Parallel and Dynamic Searchable Symmetric Encryption

Seny Kamara[1] and Charalampos Papamanthou[2]

[1] Microsoft Research, senyk@microsoft.com
[2] UC Berkeley, cpap@cs.berkeley.edu

Abstract. Searchable symmetric encryption (SSE) enables a client to outsource a collection of encrypted documents in the cloud and retain the ability to perform keyword searches without revealing information about the contents of the documents and queries. Although efficient SSE constructions are known, previous solutions are highly sequential. This is mainly due to the fact that, currently, the only method for achieving sub-linear time search is the inverted index approach (Curtmola, Garay, Kamara and Ostrovsky, *CCS '06*) which requires the search algorithm to access a sequence of memory locations, each of which is unpredictable and stored at the previous location in the sequence. Motivated by advances in multi-core architectures, we present a new method for constructing sub-linear SSE schemes. Our approach is highly parallelizable and dynamic. With roughly a logarithmic number of cores in place, searches for a keyword w in our scheme execute in $o(r)$ parallel time, where r is the number of documents containing keyword w (with more cores, this bound can go down to $O(\log n)$, i.e., independent of the result size r). Such time complexity outperforms the optimal $\Theta(r)$ sequential search time—a similar bound holds for the updates. Our scheme also achieves the following important properties: (a) it enjoys a strong notion of security, namely security against adaptive chosen-keyword attacks; (b) compared to existing sub-linear dynamic SSE schemes (e.g., Kamara, Papamanthou, Roeder, *CCS '12*), updates in our scheme do not leak any information, apart from information that can be inferred from previous search tokens; (c) it can be implemented efficiently in external memory (with logarithmic I/O overhead). Our technique is simple and uses a red-black tree data structure; its security is proven in the random oracle model.

1 Introduction

Cloud storage promises high data availability, easy access to data, and reduced infrastructure costs by storing data with remote third-party providers. But availability is often not enough, as clients need privacy guarantees for many kinds of sensitive data that is outsourced to untrusted providers. For example, privacy is clearly important for medical data, enterprise data and secret government documents.

The standard approach to achieving privacy in storage systems is to encrypt data using symmetric encryption. Storage systems based on this approach provide end-to-end privacy in the sense that data is protected as soon as it leaves the client's possession. While such a solution provides strong security guarantees, it induces a high cost in terms of functionality and is therefore inadequate for storage systems that handle data at large scales. This is because after the data leaves the client's machine in encrypted form, the server cannot perform any meaningful computation on it.

To address this, one can either use general-purpose solutions (e.g., fully-homomorphic encryption [7] or oblivious RAMs [9]) or special-purpose solutions (e.g., searchable encryption). Although general-purpose solutions have advantages, including generality and stronger security properties, they are mostly of theoretical interest (e.g., recent work [19] has shown that ORAM can be relatively practical). On the other hand, special-purpose solutions like searchable encryption are practical and aim to provide a reasonable trade-off between efficiency, functionality and security.

Using a symmetric searchable encryption (SSE) scheme, a client can store a collection of encrypted documents remotely while retaining the ability to perform keyword searches without revealing any information about the contents of either the documents or the queries. There are two high-level approaches to designing reasonably efficient and secure SSE schemes. The first approach, proposed by Goh [8] and used in [3], associates to each document an encrypted data structure that can be tested for the occurrence of a given keyword. This approach naturally results in schemes with search time that is linear in n, where n the number of documents in the collection. The second approach, introduced by Curtmola et al. [6], associates an encrypted inverted index to the entire document collection. This approach yields very efficient schemes since search time is $O(r)$, where r is the number of files that contain the keyword. Note that, $O(r)$ is not only sub-linear, it is optimal. Due to its efficiency, the inverted index approach has been used in many subsequent works, including [4,11,13,20].

While the inverted index approach yields the most efficient SSE schemes to date, it has at least two important limitations. The first is that it is not well-suited to handle dynamic collections (i.e., document collections that must be updated). Although Kamara et al. [11] recently showed how to construct an encrypted inverted index that handles dynamic data, their construction is very complex and difficult to implement. In addition, the update operations reveal a non-trivial amount of information. The second limitation of the inverted index approach is that it is inherently sequential, requiring $\Omega(r)$ time even in a parallel model of computation. This is mainly because the encrypted indexes used by these constructions store data at random disk locations (for security and space efficiency), and because the associated search algorithms are adaptive in the sense that they find the next memory location to access at the currently accessed memory location. In addition, we note that even in the sequential setting, where the $O(r)$ time bound for search is optimal, these constructions can still introduce significant latency when searching for a very frequent keyword whose output list contains thousands of documents.

Our contributions. We introduce a new approach for designing SSE schemes that yields constructions with sub-linear search time but that has none of the limitations of the inverted index approach. In particular, our approach is simple, highly parallel and can easily handle updates. More precisely, for n documents indexed over m keywords and with p cores (processors) available, our construction has the following properties:

1. Searches for a keyword w run in $O((r/p)\log n)$ parallel time, where r is the number of documents containing keyword w. Note that for $p = \omega(\log n)$, our parallel search time is $o(r)$, i.e., *less than the optimal sequential search time*.[3]

[3] Taking $p = \omega(\log n)$ is very reasonable, given that even 64-core CPUs are now available, e.g., see the TILE64 processor at `http://en.wikipedia.org/wiki/TILE64`.

Table 1. Comparison of several SSE schemes, in terms of worst case parallel search time per keyword w. With n we denote the size of the documents collection, with r the number of documents containing keyword w, with m the size of the keywords space and with p the number of cores.

scheme	**dynamism**	**security**	**search time**	**index size**
Song et al. [18]	static	CPA	$O(\frac{n}{p})$	N/A
Goh [8]	dynamic	CKA1	$O(\frac{n}{p})$	$O(n)$
Chang and Mitzenmacher [3]	static	CKA1	$O(\frac{n}{p})$	$O(mn)$
Curtmola et al. [6] (SSE-1)	static	CKA1	$O(r)$	$O(m+n)$
Curtmola et al. [6] (SSE-2)	static	CKA2	$O(r)$	$O(mn)$
van Liesdonk et al. [20]	dynamic	CKA2	$O(n)$	$O(mn)$
Chase and Kamara [4]	static	CKA2	$O(r)$	$O(mn)$
Kurosawa and Ohtaki [13]	static	UC	$O(n)$	$O(mn)$
Kamara et al. [11]	dynamic	CKA2	$O(r)$	$O(m+n)$
THIS WORK	dynamic	CKA2	$O(\frac{r}{p}\log n)$	$O(mn)$

2. Updates for a document f containing q unique keywords run in $O((m/p)\log n)$ parallel time. Again, in that case, for $p = \omega((m/q)\log n)$, our parallel update time is $o(q)$, i.e., *less than the optimal sequential update time*.
3. Finally, unlike the updates supported in the works of van Liesdonk et al. [20] and Kamara et al. [11], the updates of our scheme *do not leak* information about the keywords contained in a newly added or deleted document f, apart from information that is leaked through search tokens that have been issued in the past (updates in our scheme however require one round of interaction, as in [20]). For example, if we start adding documents into our encrypted index *before* we perform *any search* (which is common in practical applications like streaming), *no* information is leaked due to these update operations.

We finally note that our scheme enjoys a strong notion of security in this context, namely security against adaptive chosen-keyword attacks (CKA2), as defined by Curtmola et al. [6] and can also be implemented efficiently in external memory (with logarithmic I/O overhead).

Our approach. Our approach is based on a new tree-based multi-map data structure we refer to as a keyword red-black (KRB) tree. As we show in Section 3, KRB trees can index a document collection in such a way that keyword search can be performed in $O(r\log n)$ sequential time and $O(\frac{r}{p}\log n)$ parallel time. In addition, a KRB tree supports efficient updates because all the information it contains about a given file f can be found and updated in $O(\log n)$ time. To construct our SSE scheme, we show how to encrypt KRB trees based on simple and efficient primitives like pseudorandom functions and permutations and a random oracle. The resulting scheme is CKA2-secure and preserves the same (asymptotic) efficiency as an unencrypted KRB tree.

Related work. The problem of searching on symmetrically encrypted data can be solved in its full generality using the work of Goldreich and Ostrovsky [9] on oblivious RAM (ORAM). In addition to handling any type of search query, this approach also provides the strongest levels of security, namely the server does not learn any information about the data or the queries—not even information inferred by the client's access pattern. This approach requires interaction and has a high overhead for the server and the client, especially for more involved functionalities like search. Lorch et al. [14] explored the notion of parallel ORAM and proposed a parallel ORAM scheme based on a binary tree approach. Finally, while a recently introduced ORAM scheme [19] has been shown to be relatively practical, it still requires $O(\sqrt{n})$ storage at the client.

Song et al. [18] were the first to explicitly consider the problem of searchable encryption and presented a non-interactive solution that with search time that is linear in the length of the data collection. Goh [8] introduced formal security definitions for SSE and proposed a construction based on Bloom filters [1] that requires $O(n)$ search time and results in false positives. Chang and Mitzenmacher [3] proposed an alternative security definition and construction also with $O(n)$ search time but without false positives. While both the schemes of Goh and of Chang and Mitzenmacher can be naively parallelized, this would require a linear (in n) number of cores.

Curtmola et al [6]. gave the first constructions (SSE-1 and SSE-2) to achieve sublinear (and in fact optimal) search time. Like previous work [8,3], SSE-1 was shown secure against chosen-keyword attacks (CKA1). In that work, it was noted, however, that CKA1-security does not suffice for practical use. To address this, the stronger notion of security against *adaptive* chosen-keyword attacks (CKA2) was proposed and a CKA2-secure SSE scheme (SSE-2) was proposed. A similar CKA2-secure scheme was also described by Chase and Kamara [4] but its space complexity is high. Finally, recent work by Kurosawa et al. [13] shows how to construct a (verifiable) SSE scheme that is universally composable (UC). While UC-security is a stronger notion of security than CKA2-security, their construction requires linear search time.

None of the above schemes are "explicitly dynamic", i.e., to handle dynamic data one must use general dynamization techniques that are relatively inefficient (except for the scheme of Goh [8] which unfortunately has linear search time). The first explicitly dynamic scheme was presented by Liesdonk et al. [20], but unfortunately that construction supports a limited number of updates and can has linear search time *in the worst case*. Recently, Kamara et al. [11] constructed an SSE scheme that is CKA2-secure, achieves optimal search time (with small constants) and is explicitly dynamic, Unfortunately, this construction leaks the tokens of the keywords contained in an updated document. Our construction avoids such leakage and, in addition, is *considerably* simpler. A complete comparison of all the schemes can be found in Table 1.

All the above solutions are either inherently sequential or admit naive parallelization. For example, in the schemes based on the inverted index approach of Curtmola et al [4,6,11,13,20], in order to retrieve the documents $d_1, d_2, \ldots, d_r$ containing keyword w, the algorithm uses the pointer to d_1 along with a special token $t(w)$ to decrypt the pointer to d_2, then it uses the pointer to d_2 and the token $t(w)$ to decrypt the pointer to d_3 until the final pointer to d_r can be decrypted. Similarly, to delete a document that contains keywords $w_1, w_2, \ldots, w_q$, the algorithm deletes the entry corresponding to w_i only after it deletes the entry corresponding to w_{i-1}. Both these procedures are sequential in nature.

Although this work focuses on the case of single-keyword equality queries, we note that more complex queries have also been considered. This includes conjunctive queries in the symmetric key setting [10]; it also includes conjunctive queries [16,2], comparison and subset queries [2], and range queries [17] in the public-key setting.

2 Preliminaries

Our construction makes use of several basic cryptographic primitives. A *private-key encryption* scheme consists of three algorithms $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that $\mathsf{Gen}(1^k; r)$ is a probabilistic polynomial-time (PPT) algorithm that takes a security parameter k and randomness r and returns a secret key K; $\mathsf{Enc}(K, m)$ is PPT algorithm that takes a key K and a message m and returns a ciphertext c; $\mathsf{Dec}(K, c)$ 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 leak 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 *pseudorandom functions* (PRF), which are polynomial-time computable functions that cannot be distinguished from random functions by any PPT adversary and *random oracles*, to which we assume all parties have black-box access. We refer the reader to [12] for formal definitions of CPA-security, PRFs and random oracles.

Keyword hash tables. In our construction we use static hash tables [5] to store some specific information for each one of the m keywords. The entries of the hash table λ are tuples $(\mathsf{key}, \mathsf{value})$, where the key key is from a domain of exponential size, i.e., from $\{0,1\}^k$ and value is an encryption of a boolean value. However, the maximum number of entries in our hash table will be polynomial in k and equal to m, the number of keywords. If, for a table λ, the key field is from $\{0,1\}^k$, and there are at most m entries in λ, then we say λ is a (k, m) hash table. For each $x \in \{0,1\}^k$, we denote with $\lambda[x]$ the value associated with key x, if key x exists.

Searchable symmetric encryption. SSE allows a client to encrypt data so that it can later generate search tokens which the server can use to search over the encrypted data and return the appropriate encrypted files.

The encryption algorithm in an SSE scheme takes as input an index δ, a sequence of n files $\mathbf{f} = (f_{i_1}, \ldots, f_{i_n})$ that have unique identifiers $\mathbf{i} = (i_1, \ldots, i_n)$,[4] and a universe of keywords $\mathbf{w} = (w_1, \ldots, w_m)$. The index δ efficiently maps a keyword $w \in \mathbf{w}$ to a set of identifiers $\mathbf{i}_w \subseteq \mathbf{i}$ that correspond to a set of files $\mathbf{f}_w \subseteq \mathbf{f}$. The encryption algorithm outputs an encrypted index γ and a sequence of n ciphertexts $\mathbf{c} = (c_{i_1}, \ldots, c_{i_n})$, corresponding to the identifiers $\mathbf{i} = (i_1, \ldots, i_n)$. We assume all the ciphertexts include the identifiers of their plaintext files. All known constructions (except for [18]) can encrypt the files $\mathbf{f}$ using any CPA-secure encryption scheme. 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,[5] so they can be stored safely at an untrusted cloud provider.

To search for a keyword w, the client generates a search token τ_s and given τ_s, γ and $\mathbf{c}$, the provider can find the subset of ciphertexts $\mathbf{c}_w \subseteq \mathbf{c}$ that contain w. Notice that the provider learns some limited information about the client's query. In particular, it knows that whatever keyword the client is searching for is contained in whichever files resulted

[4] The identifiers are chosen uniformly at random and do not reveal any information about the plaintexts they are representing, e.g., they can be the i-nodes of the underlying file system. In order not to overload the notation, file f_i has identifier i.

[5] Note that this information leakage can be mitigated by padding if desired.

in the ciphertexts $\mathbf{c}_w$. A more serious limitation of 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.

Our scheme supports parallel keyword search as well as parallel addition and deletion of files. For updates, we use 1.5 rounds of interaction (i.e., three messages between client and server). For example, to add a file f, the client generates—with the help of the server—an addition token τ_a. Given such token and γ, the server can update the index γ. The same pattern occurs for deletion. To account for this interaction, we slightly change the definition of CKA2-security for dynamic SSE which was recently presented by Kamara et al. [11].

Definition 1 (Dynamic SSE). *A dynamic SSE scheme is a tuple* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{Search}, \mathsf{UpdHelper}, \mathsf{UpdToken}, \mathsf{Update}, \mathsf{Dec})$ *of eight polynomial-time algorithms such that:*

1. $K \leftarrow \mathsf{Gen}(1^k)$*: is a probabilistic algorithm that takes as input a security parameter* k *and outputs a secret key* K*.*
2. $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(K, \delta, \mathbf{f})$*: is a probabilistic algorithm that takes as input a secret key* K*, an index* δ *and a sequence of files* $\mathbf{f}$*. It outputs an encrypted index* γ *and a sequence of ciphertexts* $\mathbf{c}$*.*
3. $\tau_s \leftarrow \mathsf{SrchToken}(K, w)$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a keyword* w *and outputs a search token* τ_s*.*
4. $\mathbf{i}_w \leftarrow \mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$*: 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* $\mathbf{i}_w \subseteq \mathbf{i}$*.*
5. $\mathsf{info}_{i,u} \leftarrow \mathsf{UpdHelper}(i, u, \gamma, \mathbf{c})$*: is a deterministic algorithm that takes as input a file identifier* i*, the update type* $u \in \{add, delete\}$*, an encrypted index* γ *and sequence of ciphertexts* $\mathbf{c}$*. It outputs helper information* $\mathsf{info}_{i,u}$ *for the specific update.*
6. $\tau_u \leftarrow \mathsf{UpdToken}(K, f_i, \mathsf{info}_{i,u})$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K*, a file* f_i *and the respective helper information* $\mathsf{info}_{i,u}$ *as output by algorithm* $\mathsf{UpdHelper}$*. It outputs an update token* τ_u *for the update type* $u \in \{add, delete\}$*.*
7. $(\gamma', \mathbf{c}') \leftarrow \mathsf{Update}(\gamma, \mathbf{c}, \tau_u)$*: is a deterministic algorithm that takes as input an encrypted index* γ*, a sequence of ciphertexts* $\mathbf{c}$ *and an update token* τ_u*. It outputs a new encrypted index* γ' *and new sequence of ciphertexts* $\mathbf{c}'$*.*
8. $f \leftarrow \mathsf{Dec}(K, c)$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* c *and outputs a file* f*.*

We can now easily define correctness of the above dynamic SSE scheme definition.

Definition 2 (Correctness). *Let* $\mathcal{D}$ *be a dynamic SSE scheme consisting of the tuple of eight algorithms as given in Definition 1. We say that* $\mathcal{D}$ *is correct if for all* $k \in \mathbb{N}$*, for all* K *output by* $\mathsf{Gen}(1^k)$*, for all tuples* $(\delta, \mathbf{f})$*, for all tuples* $(\gamma, \mathbf{c})$ *output by one execution of* $\mathsf{Enc}(K, \delta, \mathbf{f})$ *and successive executions of* $\mathsf{Update}(\gamma, \mathbf{c}, \tau_u)$*, where* τ_u *is the update token output by* $\mathsf{UpdToken}(K, f_i, \mathsf{UpdHelper}(i, u, \gamma, \mathbf{c}))$ *for all files* f_i *and all*

$u \in \{add, delete\}$*, for all keywords* w*, for all tokens* τ_s *output by* $\mathsf{SrchToken}(K, w)$*, for all* $\mathbf{i}_w$ *output by* $\mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$*, the plaintexts* $\mathbf{f}_w = \{\mathsf{Dec}(K, c_i) : i \in \mathbf{i}_w\}$ *are all the plaintexts in* $\mathbf{f}$ *containing keyword* w*.*

Security. 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 search tokens $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_t)$ for an adaptively generated sequence of keywords $\mathbf{q} = (q_1, \ldots, q_t)$ (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 [3,6,8,11,13,20] reveal the access and search patterns.[6] 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 [4] 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. Specifically, the leakage functions we consider in this work are defined as follows:

1. $\mathcal{L}_1(\delta, \mathbf{f})$: given the index δ and the set of files $\mathbf{f}$ (along with the respective identifiers), this function outputs the number of keywords m, the number of files n, the identifiers $\mathbf{i}$ of the file and the size of each file;
2. $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$: this function takes as input the index δ, the set of files $\mathbf{f}$ and a keyword w for a search operation that took place at time t. It outputs two different types of information, namely the search pattern $\mathcal{P}(\delta, q, t)$ and the access pattern $\Delta(\delta, \mathbf{f}, w, t)$, both of which are described in the following definitions.

Definition 3 (Search pattern). *Given a search query for keyword* w *at time* t*, the search pattern* $\mathcal{P}(\delta, q, t)$ *is defined as the binary vector of length* t *with a* 1 *at location* i *if the search at time* $i \leq t$ *was for* w*; and* 0 *otherwise. Namely the search pattern reveals whether the same search was performed in the past or not.*

Definition 4 (Access pattern). *Given a search query for keyword* w *at time* t*, the access pattern* $\Delta(\delta, \mathbf{f}, w, t)$ *is defined as the identifiers in the set* $\mathbf{f}_w$ *at time* t*.*

Discussion on leakage. In our scheme (described in Section 3), if one searches for w at time t, the output of the leakage function $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$ consists of $\Delta(\delta, \mathbf{f}, w, t)$, which includes the identifiers $\mathbf{i}_w$ of the files $\mathbf{f}_w$ that contain keyword w at time t. As we add files to the collection, $\Delta(\delta, \mathbf{f}, w, t)$ is expanded with the identifiers of the new files that contain w. This is a limitation of our construction since search tokens are valid even for future documents added to the collection and effectively allow the server to use old tokens to search over newly-added documents. It is an open problem to construct *efficient* dynamic SSE schemes that do not have this limitation.[7]

[6] One exception is the construction described in [4] which leaks only the access and the intersection patterns.

[7] The scheme of Chang and Mitzenmacher [3] does not have this limitation but requires linear time search.

We note that with our construction, if one adds documents to the index *before* performing any search operations (which is common in practical applications like streaming), *no information* is leaked due to the updates. From a security point of view, this is the main difference between our scheme and the recently proposed construction of Kamara et al. [11], which always leaks information on an update. In this sense, our construction satisfies a slightly stronger notion of security since since the leakage of our updates is conditional on previous operations and does not occur unconditionally. This is also why, unlike [11], we do not use an explicit leakage algorithm for updates.

Finally, as observed in [6], 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 guarantees security if the client's queries are independent of the index and of previous results. Our scheme achieves the stronger notion of security, namely CKA2-security. In our definition of security below, we adapt the notion of CKA2-security from [11] to the setting of dynamic SSE with interactive updates. We model the interaction required by our scheme with an algorithm $\mathsf{UpdHelper}$.

Definition 5 (CKA2-security). *Let $\mathcal{D}$ be a dynamic SSE scheme consisting of the tuple of eight algorithms as given in Definition 1. Consider the following probabilistic experiments, where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator and $\mathcal{L}_1$ and $\mathcal{L}_2$ are stateful leakage algorithms:*

$\mathbf{Real}_{\mathcal{A}}(k)$: *the challenger runs $\mathsf{Gen}(1^k)$ to generate a key K. $\mathcal{A}$ outputs a tuple $(\delta, \mathbf{f})$ and receives $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(K, \delta, \mathbf{f})$ from the challenger. The adversary makes a polynomial number of adaptive queries by picking $q \in \{w, f_i\}$. If $q = w$ is a search query then the adversary receives from the challenger a search token $\tau_s \leftarrow \mathsf{SrchToken}(K, w)$. If $q = f_i$ is an update of type u, then the adversary also sends the helper information $\mathsf{info}_{i,u} \leftarrow \mathsf{UpdHelper}(i, u, \gamma, \mathbf{c})$ to the challenger and then receives from the challenger the update token $\tau_u \leftarrow \mathsf{UpdToken}(K, f_i, \mathsf{info}_{i,u})$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$: *$\mathcal{A}$ outputs a tuple $(\delta, \mathbf{f})$. Given $\mathcal{L}_1(\delta, \mathbf{f})$, $\mathcal{S}$ generates and sends a pair $(\gamma, \mathbf{c})$ to $\mathcal{A}$. The adversary makes a polynomial number of adaptive queries by picking $q \in \{w, f_i\}$. If $q = w$ is a search query then the simulator is given $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$. If $q = f_i$ is an update of type u, the simulator is given the updated output of $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$ for all keywords w that have appeared before in the adaptive queries.*[8] *The adversary also sends $\mathsf{info}_{i,u} \leftarrow \mathsf{UpdHelper}(i, u, \gamma, \mathbf{c})$ to the simulator. The simulator returns an appropriate token τ. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

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

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

[8] This will be used to simulate searches for previous tokens, the output of which (of those searches) has changed due to the update. This is the only leakage that our updates cause.

3 Our Dynamic SSE Construction

In this section we describe our parallel and dynamic construction. Let $\mathbf{f} = (f_{i_1}, \ldots, f_{i_n})$ be a sequence of documents with corresponding identifiers $\mathbf{i} = (i_1, \ldots, i_n)$ over a set of keywords $\mathbf{w} = (w_1, \ldots, w_m)$. We view each individual f_i document as a bit-string of polynomial length, i.e., $f_i = \{0, 1\}^{\mathsf{poly}(k)}$. Recall that an index δ maps a keyword $w \in \mathbf{w}$ to a set of documents identifiers $\mathbf{i}_w$.

We assume that the universe of keywords is fixed but that the number of documents can grow. In particular, we assume that the total number of keywords m is much smaller than the number of files n. We now introduce a standard (i.e., unencrypted) data structure keyword search which we refer to as a *keyword red-black* tree. KRB trees will be the basis of our dynamic SSE scheme.

The KRB tree. The KRB tree is a dynamic data structure that—similarly to an inverted index—can be used to efficiently answer multi-map queries. A KRB tree δ is constructed from a set of documents $\mathbf{f} = (f_{i_1}, \ldots f_{i_n})$ (which include the identifiers $\mathbf{i} = (i_1, \ldots, i_n)$) and a universe of keywords $\mathbf{w}$. The data structure is constructed using the following procedure, which we denote as $\mathsf{buildIndex}(\mathbf{f})$:

1. Assume a total order on the documents $\mathbf{f} = (f_{i_1}, \ldots, f_{i_n})$, imposed by the ordering of the identifiers $\mathbf{i} = (i_1, \ldots, i_n)$. Build a red-black tree T on top of $i_1, \ldots, i_n$. At the leaves, store pointers to the appropriate documents. We assume the documents are stored separately, e.g., on disk. Note that this is a slight modification of a red black tree since the tree is constructed on top of the identifiers but the leaves store pointers to the files.
2. At each internal node u of the tree, store an m-bit vector $\mathtt{data}_u$. The i-th bit of $\mathtt{data}_u$ accounts for keyword w_i, for $i = 1, \ldots, m$. Specifically, if $\mathtt{data}_u[i] = 1$, then there is at least one path from u to some leaf that stores some identifier j, such that f_j contains w_i;
3. We guarantee the above property of vectors $\mathtt{data}_u$, by computing $\mathtt{data}_u$ as follows: for every leaf l storing identifier j, set $\mathtt{data}_l[i] = 1$ if and only if document f_j contains keyword w_i. Now let u be an internal node of the tree T with left child v and right child z. The vector $\mathtt{data}_u$ of the internal node u is computed recursively as follows:
$$\mathtt{data}_u = \mathtt{data}_v + \mathtt{data}_z \,, \tag{1}$$
where $+$ denotes the *bitwise* boolean OR operation.

To search for a keyword w in a KRB tree T one proceeds as follows. Assuming that w has position i in the m-bit vectors stored at the internal nodes, check the bit at position i of node v and examine v's children if the bit is 1. When this traversal is over, return all the leaves that were reached.

The intuitive reason the KRB tree is so useful for our purposes is that it allows *both* keyword-based operations (by following paths from the root to the leaves) and file-based operations (by following paths from the leaves to the root). As we will see later, this property is useful for handling updates efficiently. We now have the following:

Lemma 1 (KRB tree data structure). *Let $\mathbf{f} = (f_{i_1}, \ldots, f_{i_n})$ be a set of n documents containing keywords from a dictionary of m keywords $\mathbf{w} = (w_1, \ldots, w_m)$. Then there exists a dynamic data structure for keyword search such that: (a) the space complexity* *of the data structure is $O(mn)$; (b) constructing the data structure takes time $O(mn)$; (c) the search time for a keyword w is $O(r \log n)$, where r is the number of documents containing w; (d) the time to insert and delete a document f is $O(q \log n)$, where q is the number of unique keywords contained in document f; (e) search and updates take parallel $O(\frac{r}{p} \log n)$ and $O(\frac{q}{p} \log n)$ time, respectively, with p processors in the concurrent-read-exclusive-write (CREW) model of parallel computation.*

Proof. Since the underlying red-black tree has space complexity $O(n)$ and we have to store at each node a bit-vector of m bits, it follows that the space complexity of the KRB tree is $O(mn)$. Now due to the property of Relation 1, given the document collection $\mathbf{f}$ one can start building the data structure following a postorder traversal. Since a postorder traversal visits $O(n)$ nodes and the time spent at each node is $O(m)$ (to compute the OR of two m-bit vectors), the time required for constructing the data structure is $O(mn)$.

Recall that sequential search for a keyword w (corresponding to position i of the m-bit vectors) proceeds as follows: while the bit at position i of node v is 1, examine v's children. Therefore the search procedure will traverse as many paths as the documents containing keyword w, namely r paths. Since the maximum height of the red-black tree is maintained to be $O(\log n)$ [5], the search time is $O(r \log n)$.

The parallel search is executed as follows. Let $0, 1, \ldots, p-1$ be the processors that are available. Processor 0 queries the root r of the tree for a specific keyword. If the search is to be continued in both the subtrees T_u and T_v of the processor's children u and v, processor 0 continues with one subtree (say T_u) and assigns the other subtree T_v to be explored by another processor. The same algorithm is recursively applied for nodes u and v. However, if at some point during the search no more processors are available (i.e., all p processors are working—this test can be achieved with concurrent read of the same data structure by all processors, that is why we require CREW model), the current processor simply selects one of its two possible children to continue, marks the other child c as "unexplored" and pushes c into a local stack of unexplored nodes so that it can be explored later. All p processors terminate their execution in $O(\log n)$ time, outputting p documents containing the queried keyword. In the second round, each processor i starts over by popping (and removing) a node c from the processor's local stack and by resuming the search from that node c. Again, during that round, each processor pushes nodes it cannot explore into its local stack. At the end of the second round, at least another p documents are retrieved in $O(\log n)$ time. Eventually, after r/p rounds of logarithmic time, all documents are retrieved (and all the local stacks are guaranteed to be empty). Therefore, search executes in parallel $O(\frac{r}{p} \log n)$ time.

The sequential update time follows from the complexity of the update of the red-black tree [5]. Since it involves bit operations between q independent vector positions, it can be implemented in $O(\frac{q}{p} \log n)$ parallel time. This completes the proof. □

We now show the following corollary, establishing the number of processors required for improving the *optimal* sequential performance:

Corollary 1 (Number of processors). *With $\omega(\log n)$ processors available, parallel searches take $o(r)$ time and parallel updates take $o(q)$ time in a KRB tree.*

KRB-based dynamic SSE. We now describe in detail our KRB-based parallel and dynamic SSE construction. Let $\mathbf{f} = (f_{i_1}, \ldots, f_{i_n})$ be the set of documents and $\mathbf{w} = (w_1, \ldots, w_m)$ be the set of keywords. We use the following cryptographic primitives:

1. A pseudo-random function $\mathtt{G} : \{0,1\}^k \times \{w_1, \ldots, w_m\} \rightarrow \{0,1\}^k$;
2. Another pseudo-random function $\mathtt{P} : \{0,1\}^k \times \{w_1, \ldots, w_m\} \rightarrow \{0,1\}^k$;
3. A random oracle $\mathtt{H} : \{0,1\}^k \times \{0,1\} \rightarrow \{0,1\}$.

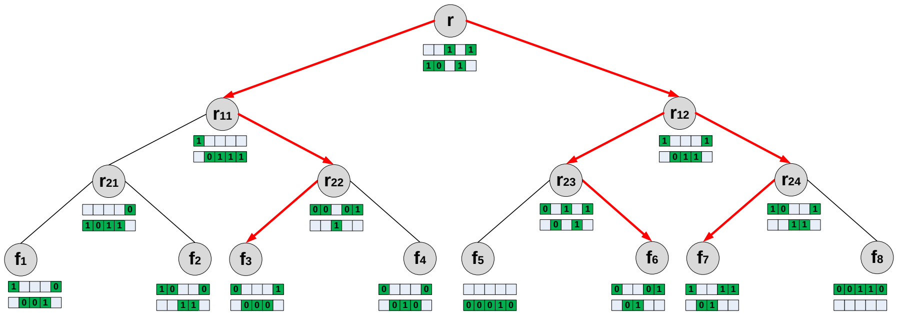

Fig. 1. The construction of a dynamic symmetric searchable encryption (DSSE) scheme using the KRB tree data structure, for a collection of $n = 8$ documents indexed over $m = 5$ keywords. Note that for each node v we store two vectors. The encryption of the actual bit of position i at node v is stored to either hash table λ_{0v} or hash table λ_{1v}, depending on the output of the random oracle. The red arrows indicate the search for keyword 5, returning documents f_3, f_6, f_7. Note that the two searches displayed can be parallelized.

We now describe the algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{Search}, \mathsf{UpdHelper}, \mathsf{UpdToken}, \mathsf{Update}, \mathsf{Dec})$ of the DSSE scheme from Definition 1 in detail:

Algorithm $K \leftarrow \mathsf{Gen}(1^k)$: Generate three random k-bit strings K_1, K_2 and r. Instantiate one private-key CPA-secure encryption scheme for encrypting documents by calling $K_3 \leftarrow \mathcal{E}.\mathsf{Gen}(1^k; r)$. Set $K := (K_1, K_2, K_3)$;

Algorithm $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(K, \delta, \mathbf{f})$: Let $\delta \leftarrow \mathsf{buildIndex}(\mathbf{f})$ be a KRB tree and proceed as follows:

1. Instantiate a second private-key CPA-secure encryption scheme $\mathcal{R}$. Derive a secret key SK_i per keyword w_i by calling $\mathsf{SK}_i = \mathcal{R}.\mathsf{Gen}(1^k; \mathtt{G}_{K_2}(w_i))$, for $i = 1, \ldots, m$.
2. For $1 \leq j \leq n$, let $c_{i_j} \leftarrow \mathcal{E}.\mathsf{Enc}(K_3, f_{i_j})$, outputting a vector of ciphertexts $\mathbf{c}$;
3. Store $\mathbf{c}$ on disk (note that the identifiers $\mathbf{i}$ remain the same) and then delete $\mathbf{f}$. For every node v of the KRB tree T, that has identifier $\mathtt{id}(v)$, do the following: **(a)** Instantiate two (k, m) keyword hash tables λ_{0v} and λ_{1v}—see Section 2 for the definition of a (k, m) keyword hash table. Store λ_{0v} and λ_{1v} at v; **(b)** For every $i = 1, \ldots, m$, set $\lambda_{\mathsf{b}v}[\mathtt{P}_{K_1}(w_i)] \leftarrow \mathcal{R}.\mathsf{Enc}(\mathsf{SK}_i, \mathtt{data}_v[i])$, where $\mathsf{b} = \mathtt{H}(\mathtt{P}_{K_1}(w_i), \mathtt{id}(v))$ is a bit computed as the output of a random oracle and $\mathtt{data}_v$ is the vector at node v in T; **(c)** Store a random string at $\lambda_{|1-\mathsf{b}|v}[\mathtt{P}_{K_1}(w_i)]$ (namely at each node v, the bit b

dictates which hash table (either λ_{1v} or λ_{0v}) contains the actual entry for keyword w_i); **(d)** Delete vector $\mathtt{data}_v$.

4. Output $\gamma := T$ and $\mathbf{c} := (c_{i_1}, \ldots, c_{i_n})$.

Algorithm $\tau_s \leftarrow \mathsf{SrchToken}(K, w_i)$**:** Call $\mathcal{R}.\mathsf{Gen}(1^k; \mathtt{G}_{K_2}(w_i))$ to output the secret key SK_i and output the search token $\tau_s := (\mathtt{P}_{K_1}(w_i), \mathsf{SK}_i)$;

Algorithm $\mathbf{i}_w \leftarrow \mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$**:** Parse τ_s as (τ_1, τ_2). Call $\mathsf{search}(r)$, where r is the root of the KRB tree T. Let v and z be the left and right child of a node u respectively. Algorithm $\mathsf{search}(u)$ is recursively defined as follows:

1. Output a bit $\mathsf{b} = \mathtt{H}(\tau_1, \mathbf{id}(u))$ and compute $\mathsf{a} = \mathcal{R}.\mathsf{Dec}(\tau_2, \lambda_{\mathsf{b}u}[\tau_1])$;
2. If $\mathsf{a} = 0$, return;
3. If u is a leaf, set $\mathbf{c}_w := \mathbf{c}_w \cup c_u$, where c_u is the ciphertext corresponding to file identifier u (and also stored at node u). Else call $\mathsf{search}(v)$ and $\mathsf{search}(z)$.

Output $\mathbf{c}_w$ (note here that, by Lemma 1, this algorithm can be parallelized to execute in $O(\frac{|\mathbf{c}_w|}{p} \log n)$ time, where p is the number of processors).

Algorithm $\mathsf{info}_{i,u} \leftarrow \mathsf{UpdHelper}(i, u, \gamma, \mathbf{c})$**:** The update u in this algorithm refers to the document f_i and is either an insertion of document f_i or a deletion of document f_i (with identifier i). To compute the information $\mathsf{info}_{i,u}$, the algorithm performs the structural update[9] on the KRB tree T. Note that in order to perform the structural update, no access to the actual content of the documents is required, since such an update is only based on the identifier i. The information $\mathsf{info}_{i,u}$ consists of the portion $T(u)$ of the KRB tree T that is *accessed* during the update. In other words $T(u)$ suffices to perform the update, in absence of the rest of the tree $T - T(u)$. Moreover, the size of $T(u)$ is $O(m \log n)$ in the worst case, given that a red-black tree update takes $O(\log n)$ time in the worst case (see Lemma 1) and therefore it can "touch" as many bits (multiplied by the size m of the encrypted vectors that are stored at the tree nodes). To give an example, in Figure 1, $\mathsf{info}_{3,u}$, where u is "deletion of file f_3", contains the nodes r_{22}, r_{11} and r (along with the encrypted vectors stored at them).

Algorithm $\tau_u \leftarrow \mathsf{UpdToken}(K, f_i, \mathsf{info}_{i,u})$**:** If the update u is an insertion of document f_i, compute first an encryption $c_i \leftarrow \mathcal{E}.\mathsf{Enc}(K_3, f_i)$ of the added document. Now let $\mathsf{info}_{i,u}$ contain the specific portion $T(u)$ of the KRB tree, as returned by $\mathsf{UpdHelper}$. Perform the structural update on $T(u)$ and let $T'(u)$ be the new subtree after the update. Every node v of $T'(u)$ that has new/modified ancestors (compared to its structure in $T(u)$) must also change its encrypted local information, since this is always computed as a function of its ancestors (in a sense, its encrypted local information is going to be recomputed and rerandomized). Also it changes its identifier from $\mathbf{id}(v)$ to $\mathbf{id}(v')$. Specifically, for every such node $v \in T'(u)$ we do the following:

1. Instantiate two *new* (k, m) keyword hash tables with random entries λ_{0v} and λ_{1v}. Store λ_{0v} and λ_{1v} at v;

[9] The structural update involves the necessary rotations that are performed during an update of a red-black tree, so that its height can be maintained to be logarithmic. The details of such operations are described in the book by Cormen, Leiserson, Rivest and Stein (CLRS) [5].

2. Update the new hash tables by setting $\lambda_{\mathtt{b}v}[\mathtt{P}_{K_1}(w_i)] \leftarrow \mathcal{R}.\mathsf{Enc}(\mathsf{SK}_i, \mathtt{data}_v[i])$ $(i = 1, \ldots, m)$, where $\mathtt{b} = \mathtt{H}(\mathtt{P}_{K_1}(w_i), \mathbf{id}(v'))$ is a bit computed as the output of the random oracle and $\mathtt{data}_v$ is the *updated* vector due to the update;
3. Output $\tau_u := (T'(u), c_i)$.

Algorithm $(\gamma', \mathbf{c}') \leftarrow \mathsf{Update}(\gamma, \mathbf{c}, \tau_u)$: On input τ_u, just *copy* the new information $T'(u)$ to the already structurally updated KRB tree T (note that T was structurally updated by the $\mathsf{UpdHelper}$ algorithm) and output the new encrypted index γ' and the new set of ciphertexts $\mathbf{c}'$.

Algorithm $f_i \leftarrow \mathsf{Dec}(K, c_i)$: Output the plaintext $f_i := \mathcal{E}.\mathsf{Dec}(K_3, c_i)$.

Correctness, security and main result. In this section we prove that our scheme is correct (Lemma 2) and secure (Lemma 3). Then we give the final result (Theorem 1).

Lemma 2 (Correctness). *The dynamic searchable symmetric encryption scheme presented above is correct according to Definition 2.*

Proof. Note that the Update algorithm modifies the encrypted KRB tree in a way that Relation 1 is satisfied, since the correct value of the bit at each node v is stored as indicated by the random oracle. Since the Search algorithm will query the same random oracle, it follows that when searching for a keyword w, it will follow the tree paths that lead to the correct set of documents.

Lemma 3 (Security). *The dynamic searchable symmetric encryption scheme presented above is $(\mathcal{L}_1, \mathcal{L}_2)$-secure in the random oracle model and according to Definition 5 (CKA-2 security), where $\mathcal{L}_1$ leaks the number of the keywords, the number of the documents, the identifiers of the documents and the size of each document; and $\mathcal{L}_2$ leaks the search pattern and the access pattern, as defined in Definitions 3 and 4.*

The proof of Lemma 3 can be found in the Appendix. We now state our final theorem.

Theorem 1 (Parallel and dynamic SSE scheme). *There exists a dynamic searchable symmetric encryption scheme for a collection of n documents indexed over a set of m keywords such that: (a) it is correct according to Definition 2 and secure according to Definition 3 and in the random oracle model; (b) searches for keywords w can be performed in sequential $O(r \log n)$ time or parallel $O(\frac{r}{p} \log n)$ time in the CREW model of parallel computation, where r is the number of documents containing w and p is the number of processors available; (c) additions or deletions of a document f_i can be performed in sequential $O(m \log n)$ time or parallel $O(\frac{m}{p} \log n)$ time, with one interaction, and have $O(m \log n)$ communication complexity; (d) the encrypted data structure γ has size $O(mn)$ and the local space needed is $O(1)$.*

Proof. Correctness follows from Lemma 2, security follows from Lemma 3 and most complexities (including the parallel ones) follow from Lemma 1. Note that the update complexity is not $O(q \log n)$, as in Lemma 1—it is $O(m \log n)$, since we do not want to reveal *which keywords are contained in the file of the update*. Also, our scheme has interactive updates due to the algorithm $\mathsf{UpdHelper}$ in the definition. □

In the following corollary we give the minimum number of processors required so that the parallel operations of our scheme outperform the *optimal* sequential complexity.

Corollary 2 (Number of processors). *With $\omega(\log n)$ processors available in the above scheme, parallel searches take $o(r)$ time. Similarly, with $\omega(\frac{m}{q}\log n)$ processors available, parallel updates take $o(q)$ time.*

4 Extensions and Optimizations

Improving the space complexity. We note here that the space complexity of our encrypted KRB tree is $O(mn)$, where m is the number of the keywords and n is the number of documents. Since $m \ll n$ in practical scenarios, the space complexity can be kept low in practice. However, one way to reduce the size of the data structure is to use a tree of depth 2 with internal nodes of degree $O(\sqrt{n})$. Similarly to KRB trees, one can store m-bit vectors at the internal nodes of such a tree and perform search, add and delete operations in the same manner. Encryption of these trees can also be handled with the same algorithm that encrypts KRB trees. The space complexity of this structure is $O(n + m\sqrt{n})$, which is $O(n)$ as long as $m \leq \sqrt{n}$. Note however that this construction increases the search time and communication complexity (for updates) to $O(r\sqrt{n})$, where r is the number of the documents that contain w.

Supporting I/O-efficient search. In the case of very large indexes that cannot fit into main memory[10], our approach can be implemented in an I/O-efficient way by using a B-tree instead of a red-black tree. With such a structure, search always requires $\log_B n$ I/Os. Note that after the disk page has been loaded into main memory, the encrypted search in main memory can be parallelized. In order now to store the B siblings of a B-tree node in a single disk page of x bits, one has to choose B such that $2Bm \leq x$. This is because at each node of the B-tree we need to store two encrypted m-bit vectors, where m is the size of the universe of keywords.

Verifiability of encrypted searches. Our model assumes an adversary that is curious but *honest*. However, we note that our scheme can be potentially extended to support verifiability of results in a scenario where the adversary is malicious. This could be achieved by turning our KRB tree into a hash-based KRB tree (e.g., using Merkle hash trees [15]) and maintaining hashes at the internal nodes of the KRB tree. For verifying the result the client could access a logarithmic number of hashes and verify the search computation step-by-step, in logarithmic time (for that, the client also needs to store and update the root hash of the tree). In this way, a client could be assured that no encrypted documents have been omitted from the results, unless the adversary is able to break the collision resistance of the hash function. We defer a more formal description of such a scheme and a proof of security to future work.

Acknowledgments

The second author was supported by Intel through the ISTC for Secure Computing. Part of this work was performed while the second author was interning at Microsoft Research. The authors would like to thank Elaine Shi, Dawn Song, Emil Stefanov and Tom Roeder for useful discussions.

[10] E.g., Amazon Common Crawl Corpus, `http://aws.amazon.com/datasets/41740`.

References

1. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference (TCC)*, pages 535–554, 2007.
3. Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS)*, pages 442–455, 2005.
4. M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 577–594, 2010.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
6. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Computer and Communications Security (CCS)*, pages 79–88, 2006.
7. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Symposium on Theory of Computing (STOC)*, pages 169–178, 2009.
8. E.-J. Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.
9. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
10. P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Applied Cryptography and Network Security (ACNS)*, pages 31–45, 2004.
11. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Computer and Communications Security (CCS)*, pages 965-976, 2012.
12. J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.
13. K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In *Financial Cryptography (FC)*, pages 285–298, 2012.
14. J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Toward practical private access to data centers via parallel ORAM. *IACR Cryptology ePrint Archive*, 2012:133, 2012.
15. R. C. Merkle. A digital signature based on a conventional encryption function. In *International Cryptology Conference (CRYPTO)*, pages 369–378, 1987.
16. D. Park, K. Kim, and P. Lee. Public key encryption with conjunctive field keyword search. In *Workshop on Information Security Applications (WISA)*, pages 73–86, 2004.
17. E. Shi, J. Bethencourt, T. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy (SSP)*, pages 350–364, 2007.
18. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Security and Privacy (SSP)*, pages 44–55, 2000.
19. E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. In *Network and Distributed System Security Symposium (NDSS)*, 2012.
20. P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management (SDM)*, pages 87–100, 2010.

Appendix

Proof sketch of Lemma 3. We describe a simulator $\mathcal{S}$ that interacts with an adversary $\mathcal{A}$ in an execution of an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ experiment as described in Definition 5. Given the leakage $\mathcal{L}_1(\delta, \mathbf{f})$, it constructs $(\gamma, \mathbf{c})$ as follows. It simulates the encrypted files $\mathbf{c} = (c_{i_1}, \ldots, c_{i_n})$ using the simulator $\mathcal{S}_{\mathcal{E}}$, which is guaranteed to exist by the CPA-security of $\mathcal{E}$, together with the value n and the size of each file (both of which are included in the leakage function). To simulate γ, it constructs a red black tree T using the identifiers

$\mathbf{i} = (i_1, \ldots, i_n)$ included in the leakage function. It then picks m random addresses $\texttt{addr}_i \in \{0,1\}^k$ and generates m keys $(\mathsf{SK}_1, \ldots, \mathsf{SK}_m)$ for $\mathcal{R}$ to be used for encrypting the entries of the bit-vectors at the internal nodes of T. Note that m is also included in the leakage function.

Now, for every node v of the tree T the simulator sets up two (k, m) keyword hash tables Λ_{0v} and Λ_{1v} as follows: for every $i = 1, \ldots, m$, the simulator stores (a) either an encryption of a "0" at $\Lambda_{0v}[\texttt{addr}_i]$ and an encryption of a "1" at $\Lambda_{1v}[\texttt{addr}_i]$ or, (b) an encryption of "1" at $\Lambda_{0v}[\texttt{addr}_i]$ and an encryption of a "0" at $\Lambda_{1v}[\texttt{addr}_i]$. This is decided, for every node, by flipping a coin. To encrypt bit $bit_i \in \{0,1\}$ at position $\texttt{addr}_i$, the simulator uses $\mathcal{R}$, outputting the ciphertext $\mathcal{R}.\mathsf{Enc}(\mathsf{SK}_i, bit_i)$ for all $i = 1, \ldots, m$. The simulator also stores this information locally: For each node v of T it stores a vector $\texttt{state}_v$ such that $\texttt{state}_v[i] = j \in \{0,1\}$ if and only if the encryption of "1" was stored at vector Λ_{jv}, for all $i = 1, \ldots, m$.[11] Finally, the simulator outputs the encrypted KRB tree γ to the adversary. γ consists of the red black tree T and the vectors Λ_{0v} and Λ_{1v} for every node v of T. We continue by distinguishing two cases, one for adaptive queries and one for adaptive updates. Before that we recall that given a search query for keyword w that takes place at time t, the leakage $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$ consists of the search pattern $\mathcal{P}(\delta, q, t)$ and the access pattern $\Delta(\delta, \mathbf{f}, w, t)$ (from Definitions 3 and 4):

1. *Adaptive queries*: Suppose the simulator receives a new query q for a keyword w_i. Using $\mathcal{L}_2(\delta, \mathbf{f}, w_i, t)$ the simulator knows whether this query has appeared before (due to the search pattern $\mathcal{P}(\delta, q, t)$), and if so, it outputs the same token τ_s. Else, the simulator picks an address $\texttt{addr}_i$ that has not been used before. The simulated token is $\tau_s = (\texttt{addr}_i, \mathsf{SK}_i)$. After the adversary receives the token $\tau_s = (\texttt{addr}_i, \mathsf{SK}_i) = (\tau_1, \tau_2)$ he is able to execute the search by using the algorithm $\mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$, in a way that Search is accessing *exactly* the same locations contained in the access pattern $\Delta(\delta, \mathbf{f}, w_i, t)$, therefore *always* returning the correct answer. We show, that this can be achieved by appropriate use of the random oracle. Namely, the output b of the random oracle at Step 1 of the $\mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$ algorithm (with reference to node v) is programmed by the simulator in the following manner: if v is contained in the path to an identifier contained in the access pattern $\Delta(\delta, \mathbf{f}, w_i, t)$, then b is the bit such that $\mathcal{R}.\mathsf{Dec}(\tau_2, \Lambda_{\mathsf{b}v}[\tau_1]) = 1$. Otherwise, b is the bit such that $\mathcal{R}.\mathsf{Dec}(\tau_2, \Lambda_{\mathsf{b}v}[\tau_1]) = 0$. Note that the appropriate bit can be determined by the simulator, since it is storing the state vector $\texttt{state}_v$, for every node v of the tree T.
2. *Adaptive updates*: Suppose the simulator receives a new query $q = u = f_i$ for inserting/deleting document f_i with identifier i. If u is an insertion, the simulator uses $\mathcal{S}_{\mathcal{E}}$ to simulate c_i (note that while f_i is not available to the simulator, its identifier is revealed through the updated leakage $\mathcal{L}_1(\delta, \mathbf{f})$). Now let $\mathsf{info}_{i,u}$ be the helper information that is returned during the protocol by UpdHelper. We recall that $\mathsf{info}_{i,u}$ consists of a certain subtree $T(u)$ of T, namely the portion of the red-black tree T that is *accessed* during the update. The simulator can now compute $T'(u)$ (since he knows the identifier of the new document) which is the new subtree after the update. For every node v' that changes its structure in the subtree $T'(u)$ (e.g., it obtains a

[11] Note that the simulator could also retrieve that information without storing it, by using the initial randomness he used to compute it.

new child), and has a new identifier $\mathtt{id}(v')$, the simulator does the following (similar actions taken in the setup phase of the simulation):

(a) it instantiates two (k, m) keyword hash tables with random entries $\Lambda_{0v'}$ and $\Lambda_{1v'}$ and stores $\lambda_{0v'}$ and $\lambda_{1v'}$ at v;

(b) For every $i = 1, \ldots, m$, the simulator *reinitializes* the keyword hash tables $\Lambda_{0v'}$ and $\Lambda_{1v'}$ by storing (a) either an encryption of a "0" at $\Lambda_{0v'}[\mathtt{addr}_i]$ and an encryption of an "1" at $\Lambda_{1v'}[\mathtt{addr}_i]$ or, (b) an encryption of "1" at $\Lambda_{0v'}[\mathtt{addr}_i]$ and an encryption of a "0" at $\Lambda_{1v'}[\mathtt{addr}_i]$. This is decided by flipping a coin. Again, to encrypt bit $b_i \in \{0, 1\}$ at position $\mathtt{addr}_i$, the simulator executes $\mathcal{R}.\mathsf{Enc}(\mathsf{SK}_i, b_i)$. Also the simulator updates its state $\mathtt{state}_{v'}$ accordingly.

(c) Output $\tau_u := (T'(u), c_i)$.

After the update, if the adversary searches for keywords for which it has old tokens, the simulator can again control the search by programming the random oracle appropriately since it gets the updated leakage $\Delta(\delta, \mathbf{f}, w_i, t)$ for each previous keyword for which a token was issued at time t.

It remains to show that for all PPT adversaries $\mathcal{A}$, the outputs of a $\mathbf{Real}_{\mathcal{A}}(k)$ and of an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ experiment are negligibly close. This holds for the following reasons. The keys used in the tokens and the ones used to encrypt the hash table elements are indistinguishable from real keys since they are constructed with PRFs which are indistinguishable from random functions. Therefore, the CPA-security of $\mathcal{E}$ and $\mathcal{R}$ guarantee that the adversary cannot distinguish between the real and simulated encryptions of the files and bit vectors, respectively. Finally, the answers returned by simulator to the adversary's random oracle queries are consistent and are appropriately distributed. $\square$