TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 2606 – 2623
© TÜBİTAK
doi:10.3906/elk-1901-157

Verifiable dynamic searchable encryption

Mohammad ETEMAD, **Alptekin KÜPÇÜ***
Computer Engineering, Koç University, İstanbul, Turkey

Received: 22.01.2019 • **Accepted/Published Online:** 05.05.2019 • **Final Version:** 26.07.2019

Abstract: Using regular encryption schemes to protect the privacy of the outsourced data implies that the client should sacrifice functionality for security. Searchable symmetric encryption (SSE) schemes encrypt the data in a way that the client can later search and selectively retrieve the required data. Many SSE schemes have been proposed, starting with static constructions, and then dynamic and adaptively secure constructions but usually in the honest-but-curious model.

We propose a verifiable dynamic SSE scheme that is adaptively secure against malicious adversaries. Our scheme supports file modification, which is essential for efficiently working with large files, in addition to the ability to add/delete files. While our main construction is proven secure in the random oracle model (ROM), we also present a solution secure in the standard model with full security proof. Our experiments show that our scheme in the ROM performs a search within a few milliseconds, verifies the result in another few milliseconds, and has a proof overhead of 0.01% only. Our standard model solution, while being asymptotically slower, is still practical, requiring only a small client memory (e.g., $\simeq$488 KB) even for a large file collection (e.g., $\simeq$10 GB), and necessitates small tokens (e.g., $\simeq$156 KB for search and $\simeq$362 KB for file operations).

1. Introduction

Huge amounts of data requiring storage, maintenance, and protection is generated these days by individuals and enterprises. Not all individuals and enterprises possess the physical and human resources required for data management. Hence, they choose to employ cloud storage services with numerous advantages such as reduced cost, high availability, and global access to data.

As the outsourced data is stored in a remote domain that the owner has no direct control on it, the data is encrypted to provide confidentiality. Searchable encryption enables the owner to outsource her encrypted files, and later search over them and retrieve files selectively, without the cloud service provider (CSP) learning the keyword or the contents of files. This requires a search token that is generated by the owner using her secret key. We focus on searchable symmetric encryption (SSE) for performance.

To store a collection of files, the client first determines the dictionary, which is the superset of keywords that appear in all files. Then, she builds an index, which is a data structure showing which file contains which keywords. She encrypts both the index and the files, and transfers them to the CSP. To search for the files containing a keyword, the client generates and sends a token enabling the CSP to search over the encrypted index, and find and return the corresponding encrypted files.

The efficient SSE schemes [1–10] reveal some information, such as the number and size of the outsourced files, the access pattern, which relates the set of encrypted files to the tokens (without learning the contents of

*Correspondence: akupcu@ku.edu.tr

the token or the files), and the search pattern, which indicates whether two or more of the tokens were for the same query [1]. A secure SSE scheme leaks nothing more.

Previous studies on cloud storage mainly considered efficient integrity verification [11–18]. Our goal in this paper is to achieve integrity and confidentiality simultaneously, while preserving the efficient search functionality in outsourced storage scenarios. We propose a verifiable dynamic SSE scheme that is secure against malicious servers, with the ability to efficiently add/delete/modify (parts of) encrypted files, and with security fully proven via simulation in both the random oracle model and the standard model.

1.1. Related work

The oblivious RAM (ORAM) [19] supports search on the outsourced data while hiding the access pattern. However, as the usage of ORAM (one block per access) differs from that of the SSE (many blocks per access), it cannot fully prevent access or search pattern leakage in SSE [20].

1.2. Early works and definitions

Goh [21] introduced the secure index as an efficient data structure for keyword search. The scheme is secure against chosen-keyword attacks (CKA1). Chang and Mitzenmacher [22] proposed the simulation-based notion of security. Curtmola et al. [1] stated that both CKA1 [21] and simulation-based [22] definitions are not adequate, and gave a stronger definition (CKA2).

1.3. Dynamic data

The problem in the dynamic setting is that once an update operation is performed, the server gains extra information related to previous queries. The server can learn whether or not the newly added files contain the keywords that have already been queried for (even though the keyword is encrypted and the server does not know it) [22]. Kamara et al. [3] extended the construction of Curtmola et al. [1] to provide a dynamic SSE (DSSE) scheme. They gave a security definition for DSSE that is adaptively secure against chosen-keyword attacks (CKA2), and presented the first *dynamic* CKA2-secure construction with optimal query time. Kamara and Papamanthou [5] proposed a parallel and dynamic SSE scheme. Stefanov et al. [6] proposed a dynamic SSE scheme with small leakage, but the scheme uses a structure similar to ORAM that requires redundant heavy rebuilds and hence is not suitable especially for devices with small storage. Cash et al. [23] proposed a dynamic SSE scheme following [24].

1.4. Verifiable SSE

Most existing SSE schemes [1–3, 22, 23, 25] provide security against semihonest adversaries. Kurosawa and Ohtaki [26] defined the verifiable SSE security, which is stronger than the 'adaptive semantic security', against the malicious adversaries [1]. These schemes support only static data. Although the schemes of Stefanov et al. [6] and Kamara and Papamanthou [5] can support verifiability for dynamic data, their proof sizes are very large (the whole index, in the worst case). Our scheme takes the advantages of both worlds and supports verifiability in dynamic settings. Recently, several papers have provided elegant solutions for verifiability and dynamism, with forward secrecy [27] and potentially via trusted hardware [28]. While the existing schemes perform modification as 'delete-then-add', our scheme supports it directly, thus more efficiently. Simultaneously with our work, Zhu et al. developed a similar solution [29] in the three-party model using Merkle Patricia trees, without file modification. We also note that Zheng et al. presented a verifiable dynamic scheme in the public key setting [30].

2. Background

We use $x \leftarrow X$ to show x is sampled uniformly from the set X, $|X|$ to show the number of elements of X, and $||$ for concatenation. PPT denotes probabilistic polynomial time, and k is the security parameter. By efficient algorithms we mean those with expected running time polynomial in the security parameter. A function $\nu(k) : Z^+ \rightarrow [0,1]$ is called negligible if $\forall$ *positive polynomials* p, $\exists$ *constant* k_0 s.t. $\forall\ k > k_0,\ \nu(k) < 1/p(k)$. Overwhelming probability is $\geq 1 - \nu(k)$ for some negligible function $\nu(k)$.

Hash functions take arbitrary-length strings, and generate fixed-length outputs. Let $h : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$ be a family of hash functions, where each member is identified by a $K \in \mathcal{K}$. A hash function family is collision-resistant if $\forall$ PPT adversaries $\mathcal{A}, \exists$ a negligible function $\nu(k)$ s. t. $Pr[K \leftarrow \mathcal{K}; (M, M') \leftarrow \mathcal{A}(h, K) : (M' \neq M) \wedge (h_K(M) = h_K(M'))] \leq \nu(k)$.

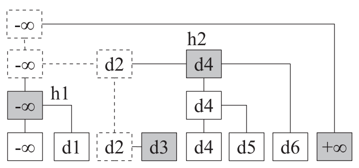
A symmetric-key encryption scheme is defined as three PPT algorithms `SKE= (Gen,Enc,Dec)` such that `Gen` is the key-generation algorithm that given the security parameter outputs a key; `Enc` is the encryption algorithm that on input the key and a message m returns the corresponding ciphertext c; and `Dec` is the decryption algorithm that takes the key and ciphertext c as input and gives the message m. We require `SKE` to be CPA-secure, which informally means that the scheme leaks no information to an adversary with access to an encryption oracle. For formal definitions, refer to [31].

For pseudo-random function (PRF), let $\texttt{GenPRF}(1^k) \in \{0,1\}^k$ be a key generation function, l the keyword length and l' the encrypted keyword length, $F : \{0,1\}^k \times \{0,1\}^l \rightarrow \{0,1\}^{l'}$ be a family of pseudorandom functions, and $F' : \{0,1\}^l \rightarrow \{0,1\}^{l'}$ be the family of all functions mapping l-bit strings to l'-bit strings. Define $F_s : \{0,1\}^l \rightarrow \{0,1\}^{l'}$ as $F_s(x) = F(s,x)$. F is a PRF family if $\forall PPT$ distinguishers $D, \exists$ a negligible function $\nu(k)$ such that: $|Pr[s \leftarrow \texttt{GenPRF}(1^k) : D^{F_s(.)}(1^k) = 1] - Pr[f' \leftarrow F' : D^{f'(.)}(1^k) = 1]| \leq \nu(k)$.

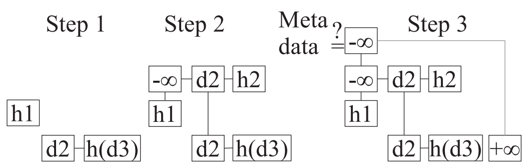
For file collections, the client owns n files $\mathbf{f} = (f_1, f_2, ..., f_n)$, each with a unique identifier $id(f_i)$. The files are encrypted as $\mathbf{c} = (c_1, c_2, ..., c_n)$ using a CPA-secure symmetric encryption scheme, where $c_i = \texttt{Enc}(K, f_i)$. The set of all unique keywords contained in the collection of files is called the dictionary, and is represented by $\mathbf{w} = \{w_1, w_2, ..., w_m\}$. We refer to the list of the files containing the keyword w as $\mathbf{f}_w$ (i.e. $\mathbf{f}_w = \{f_i : w \in f_i\}$), and to that of the encrypted files as $\mathbf{c}_w$ (i.e. $\mathbf{c}_w = \{c_i : f_i = \texttt{Dec}(K, c_i) \wedge w \in f_i\}$). The set of keywords a file contains is $\mathbf{w}_f = \{w_i : w_i \in f\}$). The files can be of any type as long as a keyword index operation for them is provided. N is the number of all keyword-file matchings.

An authenticated data structure (ADS) provides membership and nonmembership proofs for the queried data items [32]. A skip list [33] based ADS achieves logarithmic proofs [34]. Merkle hash tree [35] is another widely used ADS for static data. Each leaf node stores the hash of its associated data, and each internal node holds the hash of its children. Because the leaves are ordered, nonmembership proofs may be provided by showing two consecutive items without the queried data where it should have been. The value of the root is the digest of the ADS that is stored by the client as metadata.

Figure 1a illustrates an authenticated skip list storing six data items. '$-\infty$' and '$+\infty$' are two special values known as the left and right boundary values, respectively. The proof path of a query about d2 is drawn using the dashed lines and the parts contributing to the proof are colored. The membership proof generated



(a) An authenticated skip list.



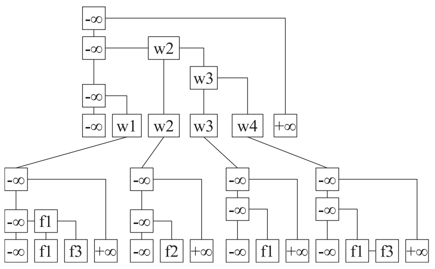
(b) Verification.

Figure 1. (a) An authenticated skip list storing six items and (b) Verifying proof of d2.

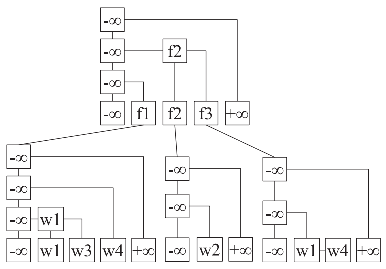
for d2 in the simplest form looks like: '$h1, d2, h(d3), h2, h(+\infty)$'. Except for the queried value that is sent in clear as part of the proof, all others are the hash values stored at the nodes in the proof path. Using the proof, the client reconstructs the required part of the ADS, and compares its digest with the one she keeps locally, as presented in Figure 1b. Any mismatch shows misbehavior of the server. Briefly, if item $d2$ did not exist (assume both nodes with label $d2$ did not exist in Figure 1a), then the nonmembership proof would have looked like '$h(-\infty), h(d1), h(d3), h2, h(+\infty)$' enabling verification of $d1$ and $d3$ being consecutive without $d2$ existing.

A hierarchical ADS (HADS) consists of multiple levels of ADSs, possibly of different types [14, 36, 37]. It relates together the relevant data stored at different levels, and can easily be distributed on multiple servers.

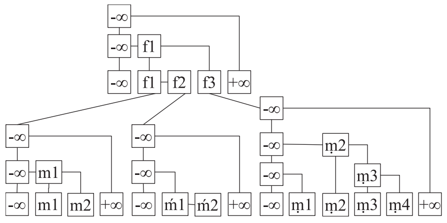
A dynamic provable data possession (DPDP) scheme provides proofs of integrity for the outsourced data, while enabling efficient updates [13]. It is possible to construct a DPDP scheme using an HADS.



(a) The forward index.



(b) The inverted index.



(c) The file index.

Figure 2. The example scenario with three files and four keywords.

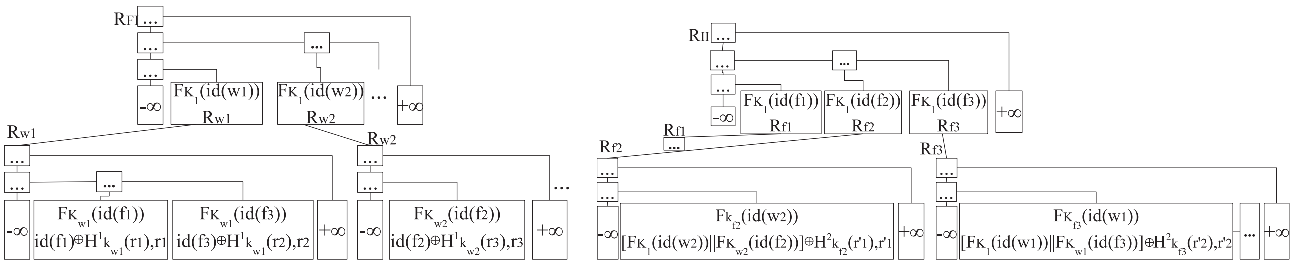
3. Construction

We use a two-level efficient HADS [37] constructed using authenticated skip lists[1] at both levels, to implement the indices. Our scheme consists of three indices: the forward index (FI) relating each keyword to the set of file identifiers it appears in, the inverted index, (II), tying each file to the set of keywords it contains, and the file index (FX), linking the DPDP structure of each file as a second-level ADS to its identifier at the first level. The ADSs have (key, value) pair-based structures.

The first level of the forward index stores the set of encrypted keyword identifiers. It is used to prove that the queried keyword does (or does not) exist in the set of stored keywords. The keys of the nodes are the outputs of a PRF on keyword identifiers, as $F_{K_1}(id(w_i))$, and the corresponding values contain R_{w_i}, which is the root of the respective second-level ADS, FI_{w_i}, storing the identifiers of the files w_i appears in.

[1]Similar ADSs, e.g., Merkle hash tree [35] or authenticated 2-3 tree [38] can also be used.

We use these second-level ADSs to prove that this set of files is exactly the set matching the keyword in a search query. For a keyword w_i, we associate two keys $K'_{w_i} = F_{K_2}(id(w_i))$ and $K_{w_i} = F_{K_3}(id(w_i))$ for hiding the identifiers of files containing w_i (i.e. the identifiers in $\mathbf{f}_{w_i}$). We use these keys to compute the (key, value) pairs (to build the FI_{w_i}) as $key_{f_j} = F_{K'_{w_i}}(id(f_j))$ and $val_{f_j} = ((id(f_j) \oplus H^1_{K_{w_i}}(r_j)), r_j)$, for all $id(f_j) \in \mathbf{f}_{w_i}$, where H^1 is a hash function modeled as a random oracle and r_j is a random value. The key_{f_j} will be used for add/delete/modify operations, and the val_{f_j} will be used during searches. A small part of the forward index construction regarding Figure 2a is shown in Figure 3a.



(a) The forward index.

(b) The inverted index.

Figure 3. A small part of real construction showing leaves storing (key, value) pairs. The upper values in boxes are the keys, and the lower ones are the values.

The inverted index, II, has a similar structure, tying a file identifier to the keywords the file contains, to support efficient deletion and modification. Without it, upon deletion of a file, the server should scan the whole forward index to find all occurrences of the file identifier; a process that is neither efficient nor private. The first-level ADS of the inverted index stores the encrypted file identifiers $F_{K_1}(id(f_j))$ as the keys, and R_{f_j} as the values at leaves. R_{f_j} is the root of the second-level ADS, II_{f_j}, storing the keywords f_j contains.

To build each II_{f_j}, first the two keys $K'_{f_j} = F_{K_2}(id(f_j))$ and $K_{f_j} = F_{K_3}(id(f_j))$ are generated. Then, they are used to compute the (key, value) pairs as $key_{w_i} = F_{K'_{f_j}}(id(w_i))$ and $val_{w_i} = ([F_{K_1}(id(w_i))||key_{f_j}] \oplus H^2_{K_{f_j}}(r_i), r_i)$, using the hash function H^2 and the random values r_i, for all keywords in f_j. They contain the information required for finding the desired nodes in the forward index efficiently (for deletion/modification). Figure 3b presents a small part of the inverted index construction corresponding to Figure 2b.

The encrypted indices above provide support for verifiability of the queries. That is, they can be used to guarantee that the file identifiers returned as a response to a search query would indeed be the real ones matching the query (see Section 3.1). Equally important is to make sure those file contents are also unmodified. Therefore, we build the file index as another two-level efficient HADS, to protect the integrity of the outsourced (encrypted) files. At the first level, an authenticated skip list is built using the file identifiers as keys, and the root of the related second-level ADSs as values. Each second-level ADS is associated with an encrypted file inside a DPDP [13] or FlexDPDP [17] instantiation to protect its integrity. The DPDP divides the file into a number of blocks, computes a tag for each block, and puts them into an authenticated rank-based skip list. Roots of these ADSs are used to construct the first-level ADS, similar to that of the inverted index[2].

The `BuildIndex` algorithm takes the files in, finds all searchable keywords among them to make the

[2]This is also similar to the directory-hierarchy extension of DPDP [13], which proves the (non-)existence of the files. Therefore, the client needs to keep only single metadata, regardless of the number of files.

dictionary **w**, and follows the above-mentioned steps to build the indices. The client stores the security keys and roots of these indices locally. The indices are then uploaded to the server along with the encrypted files.

Since the indices are encrypted, the client provides the server with the required information about each operation through tokens. The tokens depend on the client's private key, and only the client can generate such tokens. The information required for operations on the file index (according to DPDP) is also sent with each update token. For its details, we refer the reader to [13].

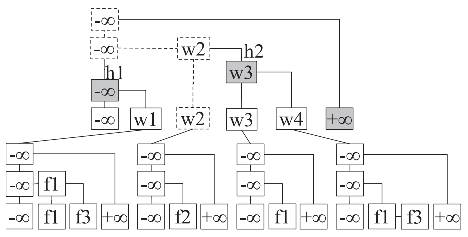
3.1. Search

3.1.1. Token

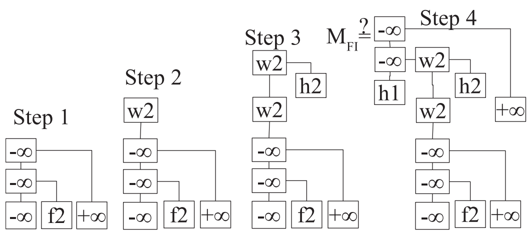
The search token carries information about a keyword w_i enabling the server to operate on the encrypted indices and find the file identifiers containing w_i. Since the forward index relates w_i to the file identifiers containing it, the token needs to include the required keys to operate on this index. Hence, we define $T_s = (F_{K_1}(id(w_i)), K_{w_i})$.

3.1.2. Server computation

Using $F_{K_1}(id(w_i))$, the server locates a leaf node in the first level of the forward index, storing the root of the respective second-level ADS. If not found, he generates a nonmembership proof for $F_{K_1}(id(w_i))$ and returns it with an empty file set to the client. If found, a membership proof for $F_{K_1}(id(w_i))$ is generated, and K_{w_i} is used to decrypt the encrypted file identifiers at leaves of the second-level ADS. Then, the server generates the integrity proofs for these file identifiers using the file index as in DPDP. Finally, all proofs together with the encrypted files are sent to the client.



(a) Search.



(b) Verification.

Figure 4. Search and verification operations on the forward index.

Example: The search token for the keyword w_2 in our example in Figure 2a is $T_s = (F_{K_1}(id(w_2)), F_{K_3}(id(w_2)))$. The first key, $F_{K_1}(id(w_2))$, specifies a path to a leaf node in the forward index, whose value will tell the server which second-level ADS to continue with, as in Figure 4a. The server generates the membership proof '$h1; w_2; h2, h(+\infty)$'. (The colored nodes contribute to the proof generation.) The second key, $F_{K_3}(id(w_2))$, is used to decrypt the leaf values of this second-level ADS to find the file identifiers.

3.1.3. Client verification

Upon receipt, the client first rebuilds the second-level ADS containing the file identifiers using the information in the proof, and uses its root to reconstruct the proof path of the first-level ADS of the forward index. Then, she compares the computed root value against the one in her local metadata, and any mismatch leads to rejection. These steps are presented in Figure 4b. If it is accepted, she compares the list of files in the answer with those given in the proof, and rejects the answer in case of any mismatch. Finally, she verifies the integrity of all the received files with the help of the DPDP part of the proof, and accepts the answer if the integrity of all files is verified.

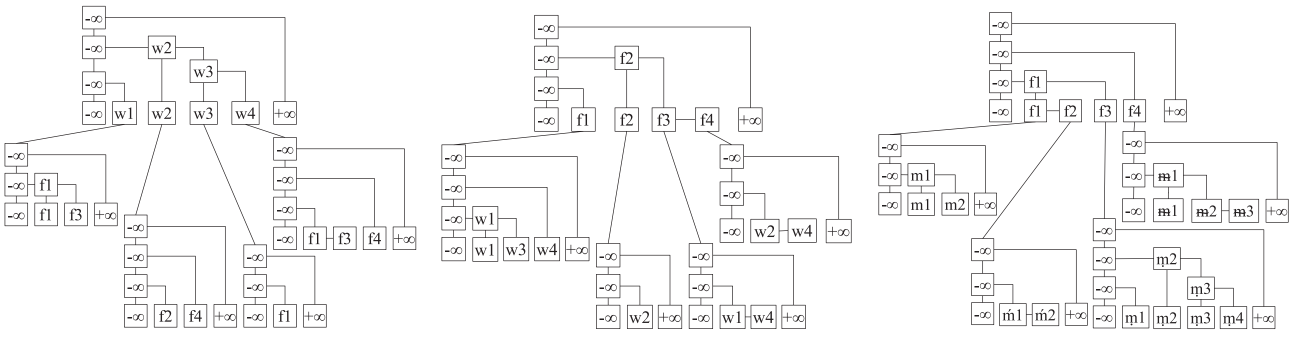

(a) The forward index. (b) The inverted index. (c) The file index.

Figure 5. The indices after adding a new file f_4 containing w_2 and w_4. The affected parts are bold.

3.2. File addition

3.2.1. Token

The add token carries information for adding the tie between the file identifier and all its keywords into the file index, updating the forward index and the inverted index accordingly. To add the file f_j containing s distinct keywords $\{w_{i_t}\}_{t=1}^{s}$, the token will look like $T_a = (F_{K_1}(id(f_j)), \{key_{w_{i_t}}, val_{w_{i_t}}, F_{K_1}(id(w_{i_t})),\ key_{f_j}^{t}, val_{f_j}^{t}\}_{t=1}^{s})$. The following example describes how the server uses this token.

Example: We add a new file f_4 with two keywords w_2 and w_4, divided into three blocks, into the example given in Figure 2. The token will be $T_a = (F_{K_1}(id(f_4)), \{key_{w_2}, val_{w_2},\ F_{K_1}(id(w_2)), key_{f_4}^{2}, val_{f_4}^{2}, key_{w_4}, val_{w_4}, F_{K_1}(id(w_4)), key_{f_4}^{4}, val_{f_4}^{4}\})$. The server first builds a second-level ADS using the key-value pairs (key_{w_2}, val_{w_2}) and (key_{w_4}, val_{w_4}), and ties its root to the first level of the inverted index using the key $F_{K_1}(id(f_4))$, as shown in Figure 5a. Then, he finds the second-level ADSs of $F_{K_1}(id(w_2))$ and $F_{K_1}(id(w_4))$ in the forward index, and adds the key-value pairs $(key_{f_4}^{2}, val_{f_4}^{2})$ and $(key_{f_4}^{4}, val_{f_4}^{4})$ into them, respectively. Finally, he instantiates a DPDP construction for this file as a second-level ADS and ties its root to the first-level ADS of the file index using the key $id(f_4)$. The changes this file addition gives rise to are identified in bold lines in Figure 5. The server sends proofs generated by each index to the client.

3.2.2. Client verification

The client first builds a second-level ADS using the keywords and the same randomness as the server, and another second-level ADS according to DPDP, and ties them to the first-level ADS of the inverted index and file index, respectively, using the proof. Moreover, she adds the file identifier into the second-level ADS of its keywords in the forward index. If all three updated indices match the proof coming from the server, the client accepts the proof and updates her metadata accordingly.

3.3. File deletion

3.3.1. Token

The token contains information required for the server to find and delete a file f_j and all relations between this file and its keywords, and update the indices accordingly. $F_{K_1}(id(f_j))$ is needed to locate the second-level

ADS of the inverted index, through which the server obtains the information required for updating the forward index. To decrypt that ADS, K_{f_j} is needed. To delete the actual (encrypted) file, $id(f_j)$ is required. Thus, the delete token is $T_d = (F_{K_1}(id(f_j)), K_{f_j}, id(f_j))$.

Example: The delete token for deleting f_3 containing keywords w_1 and w_4, from the example in Figure 3 will be $T_a = (F_{K_1}(id(f_3)), K_{f_3}, id(f_3))$. The server first locates $F_{K_1}(id(f_3))$ in the inverted index to find the respective second-level ADS, and uses K_{f_3} to compute all $H^2_{K_{f_3}}(r_i)$s to decrypt the values at its leaves to attain $(F_{K_1}(id(w_1))||key^1_{f_3})$ and $(F_{K_1}(id(w_4))||key^4_{f_3})$. Then, he locates $F_{K_1}(id(w_1))$ and $F_{K_1}(id(w_4))$ in the forward index to reach their related second-level ADSs from which he will delete $key^1_{f_3}$ and $key^4_{f_3}$, respectively. Then, he deletes the nodes storing $F_{K_1}(id(f_3))$ and $id(f_3)$ and their related second-level ADSs from the inverted and file indices, respectively, and the file $id(f_3)$. Finally, he sends the proofs to the client.

3.3.2. Client verification

The client applies all modifications locally and compares the result against what is received from the server. If the proof is accepted, she updates her local metadata accordingly.

3.4. File modification

3.4.1. Token

The modification is a combination of add and delete tokens, and causes some of the existing relation between the file and its keywords to be removed, and some new ones to be added. However, given K_{f_j} (the key of the hash function), the server can find all keywords in the file. To prevent this, we should give the server only the required $H^2_{K_{f_j}}(r_i)$s, which requires knowledge of r_is. Therefore, the client gives the server the val_{w_i}s of the keywords being deleted, receives their r_is, and prepares the required $H^2_{K_{f_j}}(r_i)$s. The token is $T_m = (F_{K_1}(id(f_j)), \{key_{w_{i_t}}, val_{w_{i_t}}, F_{K_1}(id(w_{i_t})), key^t_{f_j},\ val^t_{f_j}\}^{t_1}_{t=1}, \{key_{w_{i'_t}}, H^2_{K_{f_j}}(r_{i'_t})\}^{t_2}_{t=1}, id(f_j))$, where t_1, t_2 are the numbers of keywords added and deleted, respectively.

Even though the token treats the index modification as deletion and addition of keywords, the file operation, which is the slow and large part, is treated as a modification, according to the underlying DPDP. Previous works indeed required deleting the whole file and adding its new version from scratch.

3.4.2. Server computation

The server manages the newly-added keywords as the file addition, and those being removed as the file deletion, and sends generated proofs to the client, who performs the verification similar to the respective cases.

3.5. VDSSE in the standard model

Our scheme employs one-time pad encryption using two hash functions modeled as random oracles. Alternatively, one can use a deniable encryption scheme [39] or a noncommitting encryption scheme [40]. It is claimed that [6] and [23] can be extended to the standard model by replacing the hash functions with pseudorandom functions, but the security of the resulting schemes cannot be proven.

The one-time pad is a basic noncommitting encryption scheme: For any ciphertext, a key matching the ciphertext to the desired message can easily be found. It can effectively replace all random oracle uses.

We use two PRFs F and G to replace H^1 and H^2. Therefore, instead of storing $F_{K'_{w_i}}(id(f_j))$ and $((id(f_j) \oplus H^1_{K_{w_i}}(r_j)), r_j)$ as the (key, value) pair, we store $(l, [F_K(id(f_j)) \oplus F_{K_{w_i}}(l)])$ as the (key, value) pair at the l^{th} node of FI_{w_i}. Similarly, instead of $F_{K'_{f_j}}(id(w_i))$ and $([F_{K_1}(id(w_i))|| \ key_{f_j}] \oplus H^2_{K_{f_j}}(r_i), r_i)$, we store $(t, [F_{K_{f_j}}(id(w_{i_t}))||\langle[F_K(id(w_{i_t}))||l_t] \oplus G_{K_{f_j}}(t)\rangle])$ as the (key, value) pair at the t^{th} node of II_{f_j}.

One issue in this simple construction is that it is very likely that $id(f_1)$ appears in the first location of many second-level ADSs of the forward index. Therefore, a two-time pad attack would be possible. To prevent this type of attacks, we store $\mathbf{f}_{w_i}$ and $\mathbf{w}_{f_j}$ in permuted random order inside their second-level ADSs.

Now, in the absence of random oracles, the server should be given all $F_{K_{w_i}}(l)$ values to answer a search query (inside the search token). In a similar manner, the server needs all $G_{K_{f_j}}(t)$ values to perform a deletion, provided by the delete token. To build such tokens, the client should store locally the number of files containing each keyword w_i, cnt_{w_i}, and the number of keywords each file f_j contains, cnt_{f_j}. In essence, the tokens change, but the operations remain effectively the same as their random oracle model counterparts.

3.5.1. Search

The search token for w_i, in addition to $F_K(id(w_i))$, encompasses cnt_{w_i}-many one-time pad keys, i.e. it looks like $T_s = (F_K(id(w_i)), \ \{F_{K_{w_i}}(l)\}_{l=1}^{cnt_{w_i}})$. $F_K(id(w_i))$ specify a path in the first level of the forward index to a second-level ADS, FI_{w_i}, whose values will be decrypted using $\{F_{K_{w_i}}(l)\}_{l=1}^{cnt_{w_i}}$. The rest of the token generation and server computation are exactly as in the ROM.

3.5.2. File addition

The client extracts the keywords in the file f_j, increments their counter (cnt_{w_i}) by one, and sets $\{key_{w_{i_l}} = l\}_{l=1}^{t}$, where t is the number distinct searchable keywords in f_j. The rest of the token generation and server computation are exactly as in the ROM.

3.5.3. File deletion

Although we can give the corresponding PRF key to the server as in ROM, we cannot simulate it in our proof. Instead, the delete token for a file f_j conveys all values required for decrypting the data stored at the respective second-level ADS of the inverted index, II_{f_j}. Hence, the delete token for a file f_j with cnt_{f_j}-many distinct keywords looks like: $T_d = (F_K(id(f_j)), \{G_{K_{f_j}}(t)\}_{t=1}^{cnt_{f_j}})$. It is worth noting that since the file identifiers in the second-level ADSs of the forward index are position-dependent, removing one of them needs keeping the other positions unaffected. Thus, instead of deleting the leaf nodes containing the file identifier, we replace their contents with a NULL value showing that the node is empty. This means that deletion will not reduce the token size of later searches.

3.5.4. Modification

Since the keywords of a file are stored with the order of appearance in the related second-level ADS of the inverted index, upon modification, the client does not know their location in the related second-level ADS. As in ROM, the client first asks the server the locations of keywords being deleted, giving him the $F_{K_{f_j}}(id(w_i))$ values. Then, she generates the required $G_{K_{f_j}}(t)$ values for decrypting the encrypted data at leaves of the second-level ADS II_{f_j} corresponding to the deleted keyword. The rest of the token generation and server computation are exactly as in the ROM.

3.5.5. Optimization

If some additions and deletions are going to be performed almost simultaneously, to reduce the number of leaves with NULL values due to deletion during the modification, we can replace the new keywords with the deleted ones. However, if the number of deleted keywords is greater than that of the new keywords, the remaining deleted nodes will be replaced by NULL values. Note that this combination optimization is not a security breach, since the server already knows the (encrypted) identifiers of the deleted and added keywords in a dynamic scheme.

3.5.6. Efficiency

Previously, a search and delete token contained the keys of the hash functions. Upon receiving this key, all the server needed to do was to run the hash function with the associated r_i random values. Now, the server only stores PRF encryptions of the identifiers, without any randomness. Thus, server storage decreases a little. However, the client storage increases, as she needs to store, for each keyword and file identifier, the number of assigned file and keyword identifiers, respectively, which is $O(n+m)$, plus two extra keys used by F and G. Alternatively, the client can store the counters encrypted on the server [23] and retrieve them before performing a search. This increases communication and leakage. The search and delete token sizes also increase, depending on the number of files associated with a keyword and the number of keywords associated with a file, respectively. We show that these are very realistic numbers in practice.

4. Analysis

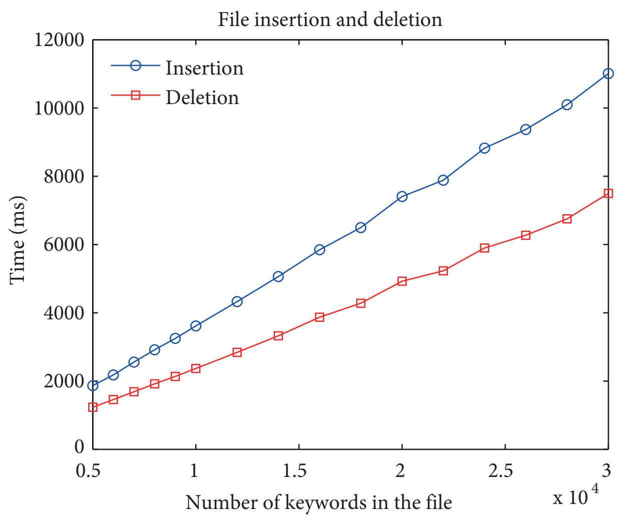
To evaluate our SSE scheme, we implemented a prototype with the two-level efficient HADS construction [36] with Flexlist [41] at both levels of the indices, in C++ using Cashlib library. All experiments were performed on a 2.50 GHz machine with 24 cores (but using a single core), with 16 GB RAM and Ubuntu 12.04 LTS operating system. The performance numbers are averages of 50 runs. We took into account only the server computation time for working on the encrypted indices, i.e. the server computation time on the files and the file index is excluded. We have two scenarios.

4.1. First scenario: small number of large documents

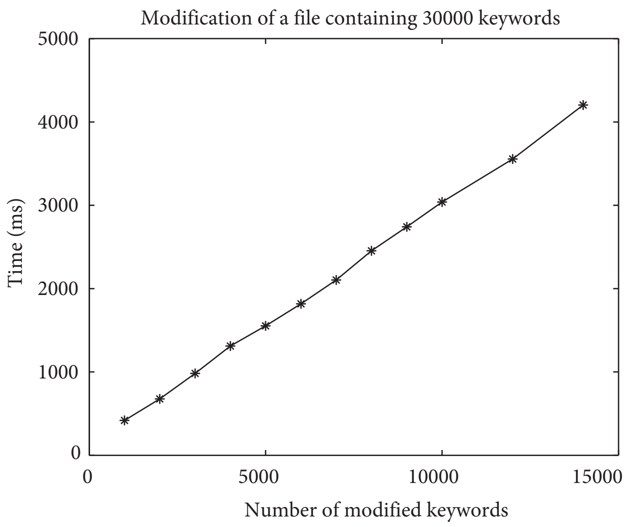
This scenario corresponds to the case that a client outsources her searchable files to a cloud server. We investigated the local storage of several accounts at Koç University and observed that there are about 1000 academic papers and ebooks, each containing 5000 to 30,000 distinct keywords, on average. There are about 100,000 distinct keywords in total. This leads to 1000 and 100,000 leaves at the first levels of the inverted index and forward index, respectively. The number of nodes of the second-level ADSs differs depending on the number of keywords each file contains, and the number of files each keyword appears in.

4.1.1. File addition and deletion

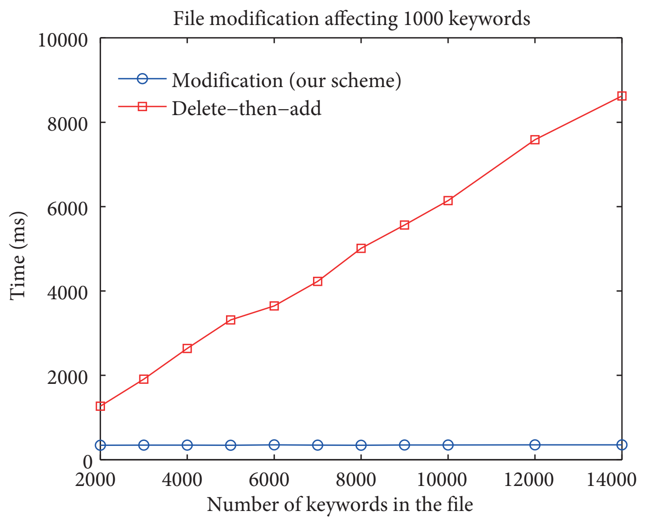
Figure 6a illustrates the addition and deletion times of a file with different number of keywords. It shows that the server performs more computation as the number of keywords in the file increases. This is expected, since the server needs to add/delete all keywords to/from the second-level ADSs of the forward index. Adding a new file with 5000 and 30,000 distinct keywords, for example, takes about 2 and 11 s, respectively.



(a) File addition and deletion.



(b) File modification.



(c) File modification.

Figure 6. File addition, deletion, and modification for the first scenario.

4.1.2. File modification

Figure 6b depicts the results of a file modification affecting a different number of keywords when the file already contains 30,000 keywords. A modification affects a set of keywords on the indices; hence, the operation time increases with the number of affected keywords. However, for a fixed number of affected keywords, the modification time is very slightly affected by the total number of keywords in the file as shown in Figure 6c; i.e. the dominant factor is the number of affected keywords.

We further compare our modification solution to first deleting the file and then adding the modified file, as required by schemes without file modification capabilities. As the Figure 6c shows, a modification on a file affecting 1000 keywords takes between 345 and 355 ms in our scheme, while the delete-then-add would require between 1500 and 9000 ms, based on the file size. This difference would get much worse if we also consider the file upload and the file index operations.

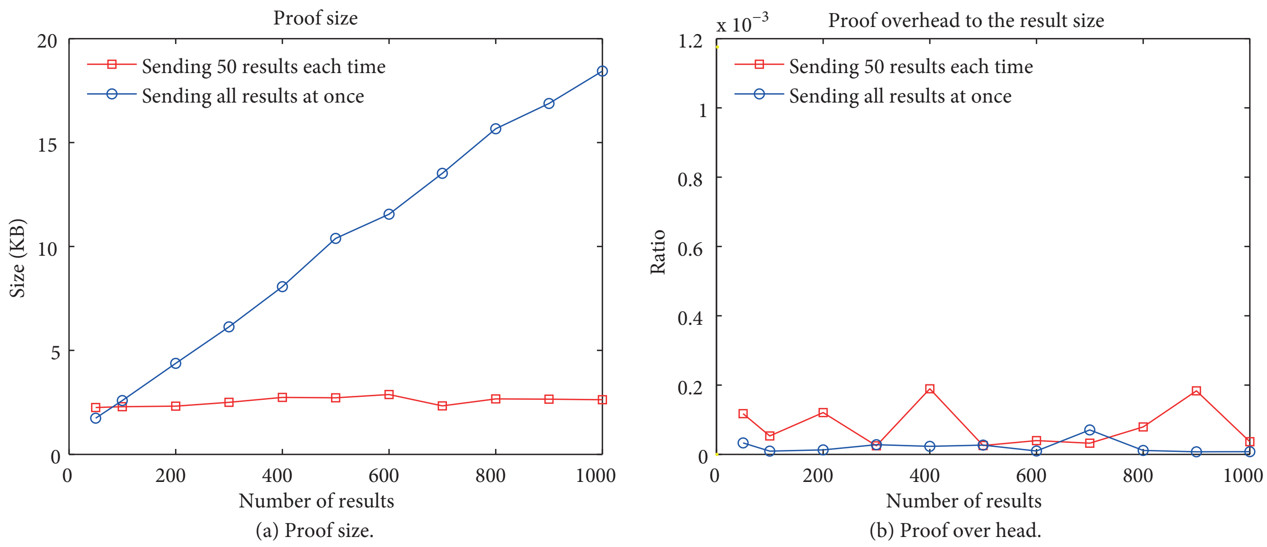


Figure 7. Proof generation and verification, and proof size and overhead to the query result.

4.1.3. File search

The server uses the first-level ADS of the forward index to generate the membership proof. Then, he decrypts all the encrypted file identifiers stored at the leaf nodes of the corresponding second-level ADS, and sends all the files along with the proof to the client. Hence, the search time does not pose a considerable variation as the number of keywords increases, and was between 7 ms and 29 ms in our tests.

4.1.4. Iterated search

The server can send all the search results together, or may send them in small groups (e.g., 50 files each time).[3] Sending the search results iteratively in groups of a small size helps the client to stop receiving more results when she is satisfied. Figure 7a compares the server proof generation time for the cases of sending all the search results together versus sending a result of size 50 files each time. It shows that the proof sizes for 100 and 1000 files in the result are about 2 and 18 KB, respectively, while that is about 2 KB for each group including 50 files. As expected, this strategy is meaningful only in scenarios where the client is expected to stop retrieval before receiving (almost) all the results.

4.1.5. Proof overhead

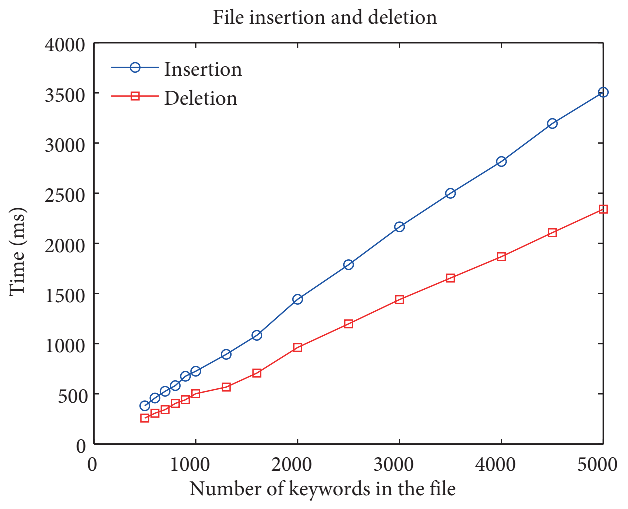
Proof overhead of our scheme is very insignificant compared to the size of resultant files being transferred, e.g., about 0.01% of the search result size as shown in Figure 7b, and can be neglected.

4.2. Second scenario: large number of small documents

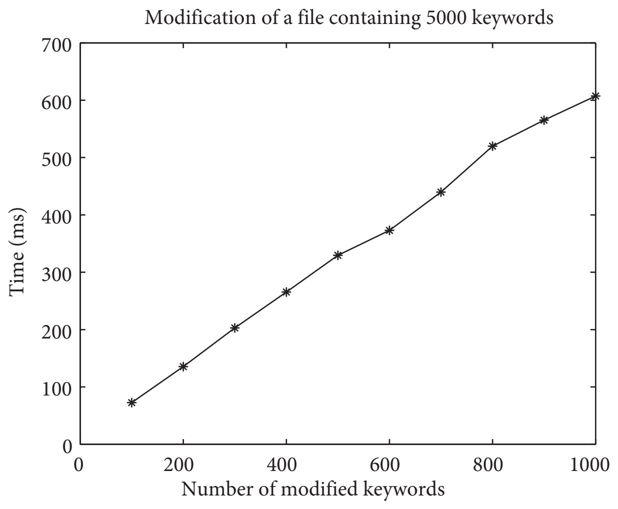
This scenario corresponds to the case that a large number (e.g., 100,000) of webpages are outsourced to an untrusted cloud server. We investigated the number of distinct keywords in different websites (e.g., bbc.com, office.com, nytimes.com, and other websites related to news, technology, and education) using the online word

[3]If the search results of all keywords are stored ranked, the client is normally interested in those with high ranks, and may not want to receive those with low ranks [8, 42].

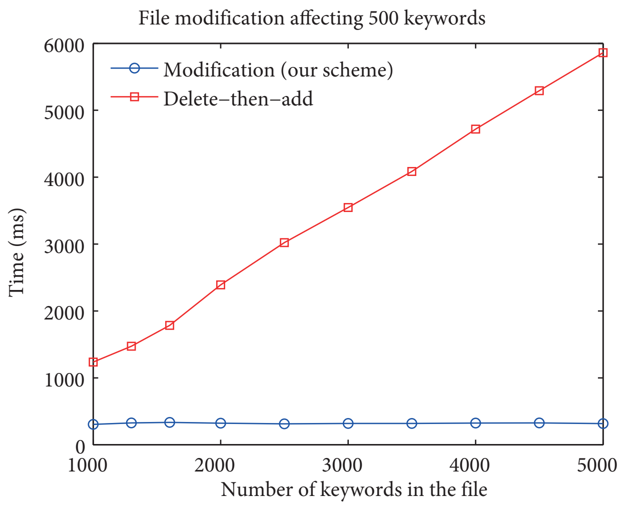
counter tool from words.contentor.com and realized that the number of distinct words in a typical webpage is generally between 100 and 5000.



(a) Fileadditionanddeletion.



(b) Filemodification.



(c) Filemodification.

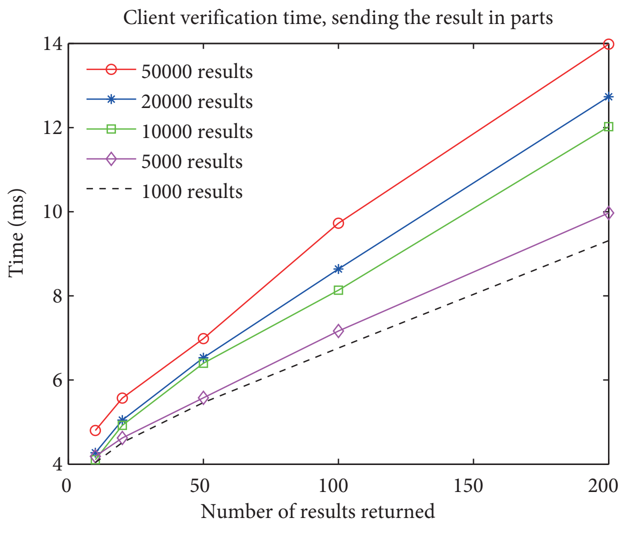
Figure 8. File addition, deletion, and modification for the second scenario.

4.3. File addition and deletion

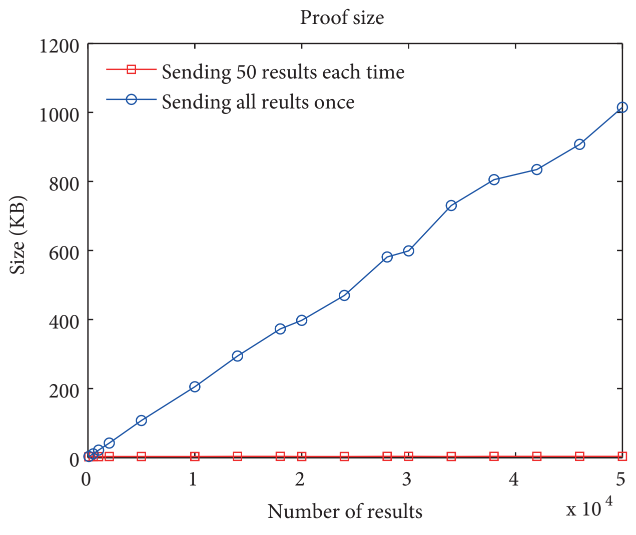
The respective times for this scenario are depicted in Figure 8a. It shows a reduction compared to Figure 6a since each file contains smaller number of distinct keywords (i.e. at most 5000). Figure 8a reveals that the addition of a new file including 500 and 5000 distinct keywords takes about 420 and 3500 ms, respectively.

4.4. File modification

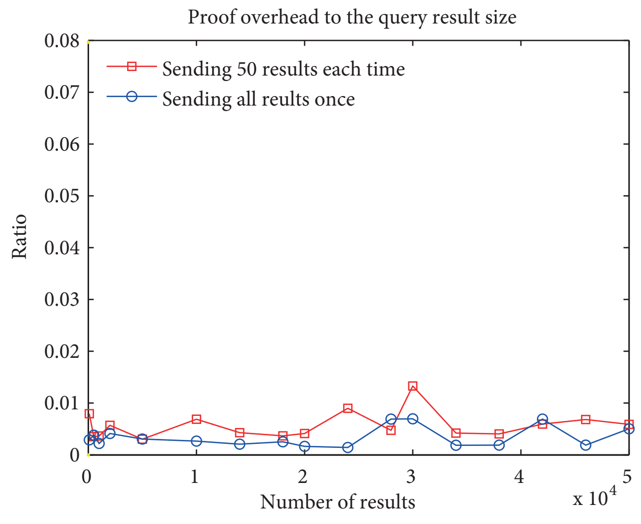
In a similar manner, the file modification shows a drop compared to the previous scenario, as shown in Figure 8b. It ranges from 80 to 600 ms when the number of keywords increases from 100 to 1000. Once again, Figure 8c shows the modification time depends primarily on the number of affected keywords, not the total number of keywords in the file. Moreover, it illustrates a modification affecting 500 keywords takes between 310 and 325 ms in our scheme, confirming the 4–20-fold efficiency gain of our modification solutions compared to the delete-then-add methods.



(a) Clientverificationtime.



(b) Proofsize.



(c) Proofoverhead.

Figure 9. The proof size and proof overhead compared to the query result.

4.5. File search

The search and verification operations behave similarly to the previous scenario. The server search time increases very slightly with the number of files in the result, and was between 5 and 11 ms.

4.6. Iterative search

This scenario resembles the search engines and similar applications, where the client receives a small part of the query results at a time (e.g., 10 per page for Google). Our server can also prepare and send a small part of the query results each time, accompanied by the corresponding membership proof. Since a consecutive part of the results are sent each time, the server uses the range query technique [37] at the second-level ADSs of the forward index, and hence the computation time changes very slightly with the number of results returned to the client. However, the client needs to verify all results she receives each time, which increases the verification time. The verification times for different sizes of the query result parts received each time are presented in Figure 9a, and are around 4 to 14 ms.

4.7. Proof overhead

Our scheme generates very small proofs, as illustrated in Figure 9b. When the whole query results are sent once, the server generates the membership proof for the keyword using the first-level ADS of the forward index, and only the file identifier existing in the corresponding second-level ADS. Therefore, the proof size increases with the number of files in the query result. When the results are sent in parts, the sizes of the proofs are independent of the number of results sent each time, and changes very slightly with the total number of the files sharing the keyword (i.e. size of the second-level ADS). When compared to the size of the files in the query result, we observe that our proofs are still insignificant in size, e.g., < 0.02 times the search result size, as shown in Figure 9c.

4.8. Standard model performance

Our scheme in the standard model requires $O(n+m)$ client storage. As the encrypted bit-length of a random file identifier and the number of keywords in a file (e.g., 128 and 20 bits, respectively) are very small compared to the size of a typical file (e.g., 10 MB), the client storage is very small compared to the outsourced files.

4.8.1. First scenario

With 1000 files ($n = 1000$) and 100,000 keywords ($m =$ 100,000), the client storage will be 101,000 $\times$ 20 = 2,020,000 bits $\simeq$247 KB, regardless of the file and keyword sizes, depending only on the number of files and keywords. The search token size is also very small, e.g., 1000 $\times$ 128 = 128,000 bits $\simeq$16 KB, when a keyword matches all 1000 files. The add token, which is the biggest one in the standard model, is of size 128 + 10,000 $\times$ (20 + 20 + 128 + 128) = 2,960,128 bits $\simeq$362 KB for a file with 10,000 keywords. The size of a delete token is about half that of an add token. The modify token depends on the number of deleted and added keywords.

4.8.2. Second scenario

For 100,000 files ($n =$ 100,000) and 100,000 keywords ($m =$ 100,000), the client storage will be 200,000 $\times$ 20 = 4,000,000 bits $\simeq$488 KB, regardless of the file and keyword sizes, depending only on the number of files and keywords. The search token size is 10,000 $\times$ 128 = 1,280,000 bits $\simeq$156 KB when a keyword matches 10,000 files. The add token size is 128 + 5000 $\times$ (20 + 20 + 128 + 128) = 1,480,128 bits $\simeq$ 180 KB for a file with 5000 keywords. The delete token size is about half that of an add token, and the size of modify token depends on the number of deleted and added keywords.

4.9. Comparison to previous work

Cash et al. [23] evaluated the performance of two of their schemes (denoted PH and 2L) on comparable hardware (see both experimental setups). In our second scenario with a large number of small files, we have around 250M (keyword, file identifier) pairs. When we compare their similar setting, we obtain the results shown in Table. Their PH solution is optimized for small result sets and performs comparable to our solution, whereas it performs much worse for large result sets. Their 2L solution, on the other hand, does not differentiate much based on the result set size, but performs worse by more than an order of magnitude, compared to our solution. This is due to the fact that in Cash et al. [23], the (keyword, file identifier) pairs are encrypted and stored at random locations that necessitate lots of disk accesses at random locations, while in our case, all file identifiers associated with a single keyword can be stored contiguously. Recently, Zhu et al. [29] generalized the work of Cash et al. [23] causing time overhead on the order of microseconds. Thus, our time comparison with Cash et al. [23] immediately applies to the comparison against the Zhu et al. [29] scheme.

Table. for comparison of search times (ms) in our scheme and [23].

# of files in the result	Cash et al. [23]		Our scheme
	2L	PH	
10	140	8	5
10,000	150	800	11

5. Conclusion

In this paper, we presented a verifiable dynamic SSE (VDSSE) scheme for outsourcing encrypted files and later retrieving them selectively, verifiably in the malicious server setting, and supporting encrypted file modification. Although our VDSSE in the standard model is asymptotically slower than its counterpart in the random oracle model, its efficiency is acceptable in practice: e.g., for 10 GB of outsourced files, on average, the client storage is $\simeq$488 KB only, and the search and add tokens are just $\simeq$156 KB and $\simeq$ 62 KB, respectively.

Acknowledgement

We acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 114E487 and 115E766, European Union COST Action IC1306, and the Science Academy BAGEP Distinguished Young Scientist Award.

References

[1] Curtmola R, Garay J, Kamara S, Ostrovsky R. Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM CCS'06; Alexandria, VA, USA; 2006. pp. 79-88.

[2] Chase M, Kamara S. Structured encryption and controlled disclosure. In: ASIACRYPT'10; Singapore; 2010. pp. 577-594.

[3] Kamara S, Papamanthou C, Roeder T. Dynamic searchable symmetric encryption. In: ACM CCS'12; Alexandria, VA, USA; 2012. pp. 965-976.

[4] Naveed M, Prabhakaran M, Gunter CA. Dynamic searchable encryption via blind storage. In: IEEE Security and Privacy (SP'14); San Jose, CA, USA; 2014. pp. 639-654.

[5] Kamara S, Papamanthou C. Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security (FC'13); Okinawa, Japan; 2013. pp. 258-274.

[6] Stefanov E, Papamanthou C, Shi E. Practical dynamic searchable encryption with small leakage. In: NDSS'14; San Diego, CA, USA; 2014. pp. 72-75.

[7] Pappas V, Krell F, Vo B, Kolesnikov V, Malkin T et al. Blind seer: A scalable private dbms. In: IEEE Symposium on Security and Privacy (SP'14); San Jose, CA, USA; 2014. pp. 359-374.

[8] Wang C, Cao N, Li J, Ren K, Lou W. Secure ranked keyword search over encrypted cloud data. In: IEEE ICDCS'10; Genova, Italy; 2010. pp. 253-262.

[9] Liu C, Zhu L, Chen J. Efficient searchable symmetric encryption for storing multiple source dynamic social data on cloud. Journal of Network and Computer Applications 2017; 86:3-14.

[10] Etemad M, Küpçü A, Papamanthou C, Evans D. Efficient dynamic searchable encryption with forward privacy. Proceedings on Privacy Enhancing Technologies 2018; 2018 (1): 5-20.

[11] Ateniese G, Burns R, Curtmola R, Herring J, Kissner L et al. Provable data possession at untrusted stores. In: ACM CCS'07; Alexandria, VA, USA; 2007. pp. 598-609.

[12] Juels A, Kaliski BS. Pors: Proofs of retrievability for large files. In: ACM CCS'07; Alexandria, VA, USA; 2007. pp. 584-597.

[13] Erway C, Küpçü A, Papamanthou C, Tamassia R. Dynamic provable data possession. In: ACM CCS'09; Alexandria, VA, USA; 2009. pp. 213-222.

[14] Etemad M, Küpçü A. Transparent, distributed, and replicated dynamic provable data possession. In: ACNS'13; Banff, Alberta, Canada; 2013. pp. 1-18.

[15] Küpçü A. Official arbitration with secure cloud storage application. The Computer Journal 2015; 58 (4): 831-852.

[16] Cash D, Küpçü A, Wichs D. Dynamic proofs of retrievability via oblivious ram. In: EUROCRYPT'13; Athens, Greece; 2013. pp. 279-295.

[17] Esiner E, Kachkeev A, Braunfeld S, Küpçü A, Özkasap Ö. Flexdpdp: Flexlist-based optimized dynamic provable data possession. ACM Transactions on Storage 2016; 12 (4): 1-44.

[18] Etemad M, Küpçü A. Generic efficient dynamic proofs of retrievability. In: ACM CCSW'16; Alexandria, VA, USA; 2016. pp. 85-96.

[19] Goldreich O, Ostrovsky R. Software protection and simulation on oblivious rams. Journal of the ACM 1996; 43 (3): 431-473.

[20] Naveed M. The fallacy of composition of oblivious ram and searchable encryption. Cryptology ePrint Archive, Report 2015/668, 2015.

[21] Goh EJ. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[22] Chang YC, Mitzenmacher M. Privacy preserving keyword searches on remote encrypted data. In: ACNS'05; New York, NY, USA; 2005. pp. 442-455.

[23] Cash D, Jaeger J, Jarecki S, Jutla C, Krawczyk H et al. Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS'14; San Diego, CA, USA; 2014. pp. 23-26.

[24] Cash D, Jarecki S, Jutla CS, Krawczyk H, Rosu M et al. Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO'13; Santa Barbara, CA, USA; 2013. pp. 353-373.

[25] Moataz T, Shikfa A. Boolean symmetric searchable encryption. In: ACM CCS'13; Alexandria, VA, USA; 2013. pp. 265-276.

[26] Kurosawa K, Ohtaki Y. Uc-secure searchable symmetric encryption. In: Financial Cryptography and Data Security (FC'12); Bonaire; 2012. pp. 285-298.

[27] Bost R, Fouque PA, Pointcheval D. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. Cryptology ePrint Archive, Report 2016/062, 2016.

[28] Ferreira B, Portela B, Oliveira T, Borges G, Domingos H et al. Bisen: Efficient boolean searchable symmetric encryption with verifiability and minimal leakage. Cryptology ePrint Archive, Report 2018/588, 2018.

[29] Zhu J, Li Q, Wang C, Yuan X, Wang Q et al. Enabling generic, verifiable, and secure data search in cloud services. IEEE Transactions on Parallel and Distributed Systems 2018; 29 (8): 1721-1735.

[30] Zheng Q, Xu S, Ateniese G. Vabks: Verifiable attribute-based keyword search over outsourced encrypted data. In: IEEE INFOCOM'14; Toronto, Canada; 2014. pp. 522-530.

[31] Katz J, Lindell Y. Introduction to Modern Cryptography. London, UK: CRC Press, 2008.

[32] Tamassia R. Authenticated data structures. In: ESA'03; Budapest, Hungary; 2003. pp. 2-5.

[33] Pugh W. Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 1990; 33 (6): 668-676.

[34] Goodrich MT, Tamassia R, Triandopoulos N. Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica 2011; 60 (3): 505-552.

[35] Merkle RC. A certified digital signature. In: CRYPTO'89; Santa Barbara, CA, USA; 1989. pp. 218-238.

[36] Etemad M, Küpçü A. Database outsourcing with hierarchical authenticated data structures. In: ICISC'13; Seoul, Korea; 2013. pp. 381-399.

[37] Etemad M, Küpçü A. Verifiable database outsourcing supporting join. Journal of Network and Computer Applications 2018; 115: 1-19.

[38] Naor M, Nissim K. Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 2000; 18 (4): 561-570.

[39] Canetti R, Dwork C, Naor M, Ostrovsky R. Deniable encryption. In: CRYPTO'97; Santa Barbara, CA, USA; 1997. pp. 90-104.

[40] Choi SG, Dachman-Soled D, Malkin T, Wee H. Improved non-committing encryption with applications to adaptively secure protocols. In: ASIACRYPT'09; Tokyo, Japan; 2009. pp. 287-302.

[41] Esiner E, Küpçü A, Özkasap Ö. Analysis and optimizations on flexdpdp: A practical solution for dynamic provable data possession. In: ICC'14; Muscat, Oman; 2014. pp. 65-83.

[42] Örencik C, Savaş E. An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking. Distributed and Parallel Databases 2014; 32 (1): 119-160.