The Turkish Journal of Electrical Engineering and Computer Sciences version is avaiable at:
https://doi.org/10.3906/elk-1901-157

Verifiable Dynamic Searchable Encryption with Boolean Search

Mohammad Etemad   Alptekin Küpçü
Koç University, İstanbul, Turkey
{metemad, akupcu}@ku.edu.tr

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. We also efficiently support Boolean search queries in such a dynamic setting, in a verifiable manner.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 [15, 13, 35, 45, 34, 52, 47, 56, 39] reveal some information, such as the number and sizes of the outsourced files, the *access pattern*, which relates the set of encrypted files to the tokens (without learning the contents of the token or the files), and the *search pattern*, which indicates whether two or more of the tokens were for the same query [15]. A secure SSE scheme leaks nothing more.

Previous studies on cloud storage mainly considered efficient *integrity* verification [1, 32, 2, 18, 49, 22, 37, 10, 50, 11]. 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.

- We present a **verifiable dynamic** searchable symmetric encryption scheme **adaptively secure against malicious adversaries**, and prove its security. Our VDSSE guarantees the authenticity and completeness of the query results and protects privacy and integrity of the documents, while enabling efficient search.
- Our scheme supports **efficient** and **provable Boolean search** for any Boolean combination of keywords (not just conjunction or disjunction alone) for dynamic data, tough it leaks all keywords in the formula (essential for verifiability in our solution).
- Unlike other schemes that only support addition and deletion operations over the encrypted files, our scheme also supports **efficient file modification**. This is an important improvement since doing a delete-then-add operation for a small change on a large file may not result in an acceptable performance.
- Our experimental results confirm the **efficiency** of our solution: A search query resulting in 500 files takes $\sim$ **4** $\boldsymbol{ms}$ to be processed by the server, with $\sim$ **11 KB proof**. Our proof overhead is only **0.01%** compared to the total size of the resultant files. Moreover, adding (deleting) a new file containing 10,000 distinct keywords takes $\sim$ 3800 (2300) ms. A modification on the file affecting 1000 keywords takes between 345 and 355 ms, depending on the file size.
- We also propose a dynamic construction secure in the **standard model** with full simulation security proof and Boolean search capability. While being asymptotically slower than its random oracle model counterpart, we argue that 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 $\simeq$ **362 KB**, respectively.

1.1 Related Work

The oblivious RAM (ORAM) [28] supports search on the outsourced data while hiding the access pattern. Recently, it is used to construct SSE schemes for hiding memory accesses and preventing leakage [29, 52, 3, 26].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 [44].

Early works and definitions. Goh [27] introduced the *secure index* as an efficient data structure for keyword search. The search time is linear in the number of files.The scheme is secure against chosen-keyword attacks (CKA1), but Bloom filters cause false positives. Chang and Mitzenmacher [12] gave constructions with $O(n)$ search time and proposed the *simulation-based* notion of security that shows the trapdoors do not leak any information about the keywords queried. Curtmola *et al.* [15] stated that both CKA1 [27] and simulation-based [12] definitions are not adequate, and gave a stronger definition (CKA2). Adaptively and non-adaptively secure schemes with optimal query time $O(d)$ were proposed, where d is the number of matching documents.

Chase and Kamara [13] introduced the concept of *controlled disclosure*, i.e., the client discloses some pieces of (encrypted) data that are necessary for the server to perform its task. They gave constructions for queries on matrices, labeled data, and graphs. The simulation-based security was used in their proofs.

Yoshino *et al.* [58] pointed to a security weakness in the existing games for SSE, and gave a new game, for database applications. But the constraints they put are hard to satisfy, and the proposed construction has a linear (in the table size) search time.

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. Consider that the scheme uses previous pseudo-random seeds and keys for the updated files. Then, 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) [12]. Using a new key for each update and using a pseudo-random function to generate keys, require a key management scheme and make the search process complicated. Curtmola *et al.* [15] gave an optimization to reduce the complexity of such key management from linear to logarithmic.Van Liesdonk [55] proposed two dynamic SSE schemes. The first scheme requires a large communication for search, and the second scheme has a limited number of updates. Kamara *et al.* [35] extended the construction of Curtmola *et al.* [15] 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 [34] proposed a parallel and dynamic SSE scheme using a *keyword red-black tree*, with $O((d/p)\log n)$ (parallel) search time, where p in the number of processors running the operation in parallel. But, the dictionary is static, and large amount of data is stored at nodes, making the proofs large. Stefanov *et al.* [52] proposed a dynamic SSE scheme with small leakage, and achieving forward privacy (i.e., current leakage is not useful for future updates). 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.* [8] proposed a dynamic SSE scheme following [9]. It stores all existing keyword-file pairs encrypted and at a random order on the server. The search time in this scheme is $O(d)$.

Verifiable SSE. Most existing SSE schemes [12, 15, 13, 35, 42, 8] provide security against semi-honest adversaries. They assume that the cloud server follows the protocol and gives the correct answers to the client while trying to learn more information about the outsourced data. A more rough adversarial setting considers an adversary that follows the protocol, but may not send back the correct set of files matching the query (attacking completeness of the result) [57, 59, 53].Kurosawa and Ohtaki [38] defined the *verifiable SSE* security against the malicious adversaries, that is stronger than the 'adaptive semantic security' [15]. These schemes support only static data. Although the schemes of Stefanov *et al.* [52] and Kamara and Papamanthou [34] 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 provided elegant solutions for verifiability and dynamism, with forward secrecy [4], potentially via trusted hardware [24]. 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 [60] 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 [59].

Boolean search. Search for Boolean combinations of keywords was an open problem for a while [12, 33]. Existing schemes in the semi-honest model [27, 42, 9, 47, 25] ask the server to perform the Boolean search and send back the files matching the query, without a proof showing that the query was executed properly and the returned files are the exact ones. *Completeness* of the search results is a problem in the malicious setting. PDSSE [34] can support Boolean search with large proofs for static dictionary. Stefanov *et al.* [52] scheme can be extended to support verifiability, but cannot handle efficient Boolean search. BSSE [42] uses randomized queries and supports general form of Boolean search in the semi-honest setting. Cash *et al.* also support efficient Boolean search operation in the semi-honest setting for static [9] and dynamic [8] data. Blind seer [47] goes one step ahead and preserves privacy of the client query. In fact, the query is sent to the server encrypted and the server returns the (encrypted) files matching the query. However, blind seer is mainly for static data and can support updates in a basic way, which is not suitable for highly dynamic environments, and works in the semi-honest setting. The malicious-client blind seer [25] considers and solves an access control deviation problem in blind seer when the client acts maliciously. We perform verifiable Boolean search in dynamic setting efficiently, though leaking all keywords in the Boolean formula, and leave the privacy-preserving, verifiable and efficient Boolean search in dynamic settings as an open problem.

Attacks. Islam *et al.* [31] combined the access pattern leakage with the prior knowledge gained from other streams to find the queried keywords. Liu *et al.* [40] exploited the search pattern leakage with the help of prior knowledge to learn information about the words being searched. Cash *et al.* [7] classified the the SSE leakages and investigated how they can be abused on different schemes.

Comparison. A comparison among the dynamic schemes is given in Table 1. It reveals that our scheme is the only dynamic scheme adaptively secure in the presence of malicious adversaries, supporting efficient and secure file modification as well. We also support Boolean search in an efficient and verifiable manner, though our leakage in the Boolean search is more than the existing work, since we support both dynamism and verifiability. While [52] and [8] are claimed to be extensible to the standard model, the security of the resulting schemes cannot be simulated as is, and needs more work. We give a full construction in the standard

Table 1: A comparison of dynamic SSE schemes. ('ROM' and 'STD' stand for random oracle and standard models, respectively. 'Honest+' means possibility of extending to the malicious setting.)

Scheme	Adv. model	Dynamism	Model	Boolean	Verifiable	File integrity
[55]	honest	Add, Delete	ROM	–	–	–
[35]	honest	Add, Delete	ROM	–	–	–
[34]	honest+	Add, Delete	ROM	–	–	–
[52]	honest+	Add, Delete	ROM	–	–	–
[47]	honest	Add, Delete	ROM	$\surd$	–	–
[8]	honest	Add, Delete	ROM	$\surd$	–	–
[25]	malicious-client	Add, Delete	ROM	$\surd$	–	–
Ours	malicious	Add, Delete, Modify	ROM & STD	$\surd$	$\surd$	$\surd$

model and prove its security via simulation. A more detailed comparison will be provided in Section 5.2.

2 Background

2.1 Preliminaries

Notation. 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. $Pr[K \leftarrow \mathcal{K}; (M, M') \leftarrow \mathcal{A}(h, K) : (M' \neq M) \wedge (h_K(M) = h_K(M'))] \leq \nu(k)$.

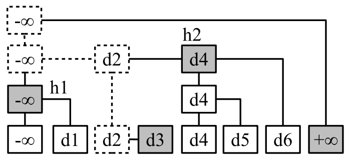
A **symmetric-key encryption scheme** is defined as three PPT algorithms `SKE= (Gen,Enc,Dec)` such that `Gen` is the key-generation algorithm that given the security parameter outputs a key; `Enc` is the encryption algorithm that on input the key and a message m returns the corresponding ciphertext c; 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 [36].

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: $|Pr[s \leftarrow \texttt{GenPRF}(1^k) : D^{F_s(\cdot)}(1^k) = 1] - Pr[f' \leftarrow F' : D^{f'(\cdot)}(1^k) = 1]| \leq \nu(k)$.

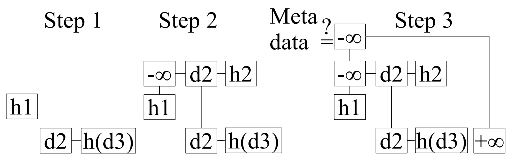
Searchable symmetric encryption enables a client to store encrypted data on the server, and later query the encrypted data, giving the server the corresponding search tokens, which are generated using the client's secret key. The server uses the search token to find and return the matching encrypted files.

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 non-membership proofs for the queried data items [54]. A skip list [48] based ADS achieves *logarithmic* proofs [30]. Merkle hash tree [41] 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, non-membership proofs may be



(a) An authenticated skip list.



(b) Verification.

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

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 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 non-membership 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 [22, 21, 23]. It relates together the relevant data stored at different levels, and can easily be distributed on multiple servers. The formal definition of HADS is provided in C for reference.

A dynamic provable data possession (**DPDP**) scheme provides proofs of integrity for the outsourced data, while enabling efficient updates [18]. Using hash functions or message authentication codes over the whole outsourced file instead, will not work as they cannot support later updates efficiently. The formal definition of DPDP is given in D for reference. It is possible to construct a DPDP scheme using an HADS.

2.2 Our Model

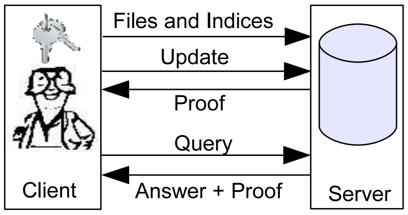


Figure 2: Our VDSSE model.

There are two parties in our Verifiable Dynamic SSE (VDSSE) model. The *client* performs the required pre-computations, builds the indices, and uploads the (encrypted) indices and files to the *server*. The server is in possession of resources and equipment required for hosting the outsourced data and answering the client requests. Later, the client asks the server to perform search, deletion, modification, or addition on the encrypted data, giving the appropriate token and other necessary information. The token contains the required information that enables the server to perform the operations on the encrypted indices, and can be generated using the secret key of the client. Other necessary information may include the new file to be added, modifications to the file, and possibly pseudo-random seeds for randomized operations. The server performs the operation, and generates and sends the answer and proof to the client. For simplicity, we assume a single-client model as presented in Figure 2.

Using the search token, for example, the server finds the set of encrypted files $\mathbf{c}_w$ that contain the keyword w, and sends back those files along with the proof that the search was indeed done as requested. The proof is *necessary* in the *malicious* setting. The client will accept the result if and only if the proof is verified.

All these operations are performed on the encrypted data, and hence, the server will not find out which keyword is searched for, or what the contents of the files are. But he will learn that all files in the set of (encrypted) files selected as the search result contain the queried keyword, as well as the fact that all other files do not. Moreover, update operations reveal which encrypted keywords are added/removed to/from which encrypted files. Hence, dynamic schemes leak more information than the static ones [35]. We allow access pattern and search pattern leakages, and precisely specify what leaks with each token.

Adversarial model. The server can act maliciously, or be subverted by the attackers to do so. He may cheat by attacking the verifiability (sending a wrong set of files to the client), or the integrity of the outsourced data (modifying the file contents), while trying to be undetected. Furthermore, he may try to obtain more information about the keywords or files than what is leaked by the tokens.

2.3 Overview of Our Solution

Problems. Memory-checking schemes can support confidentiality in addition to integrity [1, 32, 18, 5, 49, 16, 2, 20]. But, we need to solve two more problems:1) efficient search over encrypted data, and 2) *verifiability* (proving that the files returned by the server are *exactly* the ones matching the query, i.e., there is no extra or missing or corrupted file). The straightforward solutions either require employing deterministic encryption as in [51], which is vulnerable to statistical attacks leading to revealing the file contents, or transferring all the documents to the client, which is not efficient. SSE was mainly proposed as a remedy to the first problem, while it should take into account the second one as well.

The Boolean search makes the problem even worse, as it requires providing verifiability for multiple keywords efficiently. The problem is that the (encrypted) indices are built around single keywords, without any efficient algorithm for combining them and generating proofs for a Boolean combination of keywords. Existing solutions (e.g., [9, 42, 47, 25]) work in the semi-honest setting and do not support verifiability.

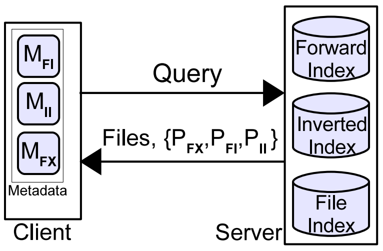


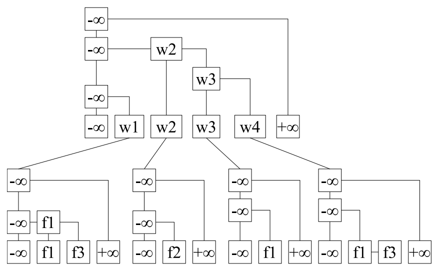
Figure 3: VDSSE architecture.

Our Solution. To be secure against malicious adversaries, an SSE scheme may employ two parts: an integrity-preserving part (i.e., a memory-checking scheme) and a verifiability-providing part (i.e., an index-based SSE scheme). The server storage in our scheme consists of three parts: the *file index* (FX) stores security information of the encrypted files and generates integrity proofs; the *inverted index* (FI) and the *forward index* (II) are the encrypted indices used for providing verifiability. The client stores the keys and digests of these indices (R_{FX}, R_{FI}, and R_{II}) as metadata for verification. The server's answer contains the requested files and the associated verifiability and integrity proofs (P_{FI}, P_{II} and P_{FX}) as shown in Figure 3.

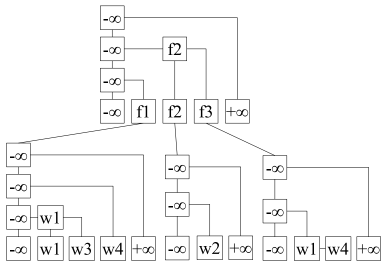
The **inverted index** is a two-level HADS tying each keyword to the set of file identifiers containing the keyword. The encrypted keywords are located in the first-level ADS, which is used to generate *proofs* showing *(non-)existence* of the queried (encrypted) keyword(s). This is a new feature not provided by most of the existing schemes. Each node storing a keyword is connected to another ADS at the second level, who stores the encrypted file identifiers containing the keyword. The proofs generated using this second-level ADS assures the client about the verifiability of the received response (i.e., no extra or missing files).

The inverted index does not suit file deletion well since the server should traverse the whole data structure to find all occurrences of the file and delete them. The **forward index** links each file identifier to the set of keywords the file contains, for efficiently locating them in the inverted index. Therefore, we build another two-level HADS that stores the file identifiers in the first level, and links them to the second-level ADSs storing the encrypted keywords each file contains. The elements in each second-level ADS carry information needed for locating the corresponding keywords in the inverted index. To delete a file, the server finds its keywords using the forward index, removes them from the forward index, and uses the provided information to efficiently find and delete the links between these keywords and the file in the inverted index.

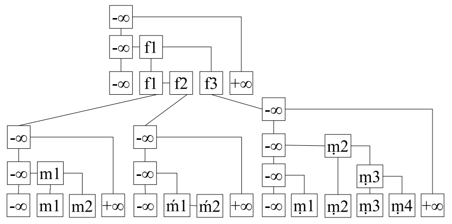
The **file index** is another two-level HADS. The first-level ADS stores the file identifiers, and provides proofs showing (non-)existence of the files. For each file, a second-level ADS is constructed as a dynamic provable data possession (DPDP) scheme [18, 20], providing its integrity. All update operations on the files are prepared and performed accordingly. For each operation, DPDP provides a proof through which we can verify that the operation is performed correctly.



(a) The forward index.



(b) The inverted index.



(c) The file index.

Figure 4: The example scenario with three files and four keywords.

These three data structures support provable operations (search/add/delete/modify) on the encrypted outsourced files. The first-level ADSs fulfill two functionalities: providing (non-)membership proofs for the queried keywords and file identifiers, and locating the corresponding second-level ADSs. A search operation accesses the forward and file indices (FI and FX), while the update operations on files (add/delete/modify) affect the forward index (II), too. This way, future search operations return correct and consistent answers.

The client stores the digests (the values stored at the root) of the three HADSs, together with the required keys (three PRF keys, one symmetric encryption key, and the DPDP key). Hence, the client storage is $O(1)$.

An illustrative example. We use an example to better understand the indices and the relations among them. Assume that there are three files and four keywords: f_1 contains (w_1, w_3, w_4), f_2 contains (w_2), and f_3 contains (w_1, w_4). The files are divided into two, two, and four blocks, respectively. Figure 4 shows a simplified representation of the indices where the file and keyword identifiers are used instead of their encrypted versions. The HADSs employed in the figure are based on authenticated skip lists.

Search. To answer a search query, the server first locates the queried keyword(s) in the first-level ADS of the inverted index (thus information about individual keywords leak). If not found, he generates and sends back a non-membership proof. If found, he generates the membership proof, and finds the respective second-level ADS(s) to extract the identifiers of all files containing the keyword(s). Then, he prepares the integrity proof for those files using the file index. Finally, he sends all proofs and files matching the query (if any) to the client. For a Boolean search, we employ an algorithm to operate via intersections, unions, or complementations on the second-level ADSs of all queried keywords and generate efficient proofs. This way, the server only needs to send the final resulting set of files to the client, even in the malicious setting. Even though the leakage is not optimal, it is a first-of-its-kind efficient solution to work in a dynamic and verifiable manner.

File addition. The server first generates a second-level ADS including all (encrypted) keywords in the file (given in the token), and ties its root to the file identifier in the first level of the forward index, as shown in Figure 7b, where a new file f_4 containing keywords w_2 and w_4 is added. Then, he adds the file identifier into the second-level ADSs of all its keywords (w_2 and w_4). The result is illustrated in Figure 7a. He also builds a second-level ADS according to DPDP and relates its root to the file identifier in the first level of the file index, as in Figure 7c (illustrated assuming that the file is composed of three blocks). Finally, he sends proofs of all these operations to the client.

File deletion. To delete a file, the server first locates the corresponding second-level ADS of the file in the forward index, using which he accesses all file keywords to delete the relations between the file identifier and all its keywords in the file index. Then, he deletes the corresponding second-level ADS of the file in both the forward index and the file index. Finally, he deletes the file itself, and sends the proofs of all three indices to the client. As an example, to delete f_3 from Figure 4, the server realizes through the related second-level ADS in the forward index, that f_3 has two keywords: w_1 and w_4. Using this information, he finds and deletes f_3 from the second-level ADSs of w_1 and w_4 in the inverted index. Then, he deletes f_3 and its second-level ADS from both the forward index and the file index.

File modification deletes some already-existing keywords and adds some new ones by operating on the forward and inverted indices as the file addition and deletion, while performing a modification on file index.

3 VDSSE: Verifiable Dynamic Searchable Symmetric Encryption

3.1 Security Description

A secure SSE scheme should be *verifiable* (i.e., satisfy file integrity and query completeness in the presence of malicious adversaries) and *private* (i.e., the adversary cannot gain more information beyond what is allowed). We define these security requirements intuitively first, and then provide formal definitions in the next section.

Verifiability. The server cannot deceive the client into accepting a wrong response. If he adds, deletes, modifies contents of some files, or sends a set of files different from the actual query result, he will be caught with overwhelming probability. We formalize this concept through a game similar to that of an ADS [30, 21].

Privacy. Ideally, *nothing* should be leaked about the keywords and the files [15, 13, 35, 52]: 1) the index and the set of encrypted files reveal no information about the original index and files, and 2) the set of adaptively-generated tokens reveals no information about the queries, the index, and the files. Almost all existing SSE schemes [27, 12, 15, 13, 35, 34, 52, 53, 8] reveal the access and search patterns for efficiency.

In fact, by uploading the encrypted index and files, the server learns at least the number of files and their sizes. Moreover, each query leaks some limited information about the files and keywords, e.g., the server will learn all encrypted files found as the result of a search query share the encrypted keyword, and whether or not different queries were for the same keyword. Dynamic schemes reveal more information [35].

We prove that both of our solutions (the random oracle model version, as well as the standard model scheme) are CKA2-secure. This is proven using a simulator that acts in an indistinguishable manner to the client, and performs operations by only knowing the leakage. Since the simulator performs the operations only knowing the leakage, it is impossible for the adversary to learn more than the leakage. Essentially, this is the same proof strategy as in the zero-knowledge simulator for the proofs.

Leakage. For each operation, the client reveals some information to the server to help him fulfill the job. Therefore, some information about the client's queries and the files containing each (queried) keyword leaks. To formalize the leakage during the system initialization and execution, we use multiple leakage functions (described in Section 5.1) each capturing the information leakage from a different viewpoint.

3.2 Formal Definitions

Definition 1 *A verifiable dynamic SSE scheme includes the following PPT algorithms:*

The following algorithms are **executed by the client**:

$(sk, pk) \leftarrow \texttt{KeyGen}(1^k)$ is an algorithm to generate a secret and public key pair (sk, pk) given the security parameter k in unary. The public key pk is shared with the server.

$(\mathcal{I}, \mathbf{c}, M) \leftarrow \texttt{BuildIndex}(sk, \mathbf{f})$ is an algorithm that given the secret key sk and a file collection $\mathbf{f}$, creates the encrypted index $\mathcal{I}$ and file collection $\mathbf{c}$ to be sent to the server, and the local metadata M.

$f = \texttt{Dec}(sk, c)$ is a deterministic algorithm that given the secret key sk and a ciphertext c, outputs the corresponding plaintext file f.

$T_s = \texttt{SearchToken}(sk, M, w)$ given the secret key sk, the metadata M, and a keyword w from the dictionary, creates a search token T_s to be sent to the server.

$T_b = \texttt{BooleanSearchToken}(sk, M, \phi(w_1, w_2, ..., w_t))$ is an algorithm that given the secret key sk, the metadata M, and the Boolean combination of keywords (using '$\wedge$', '$\vee$', '$\neg$'), computes a Boolean search token T_b that is sent to the server.

$(T_a, c) = \texttt{AddToken}(sk, M, f)$ is an algorithm that given the secret key sk, the metadata M, and a file f, creates an addition token T_a and the corresponding ciphertext c.

$T_d = \texttt{DeleteToken}(sk, M, id(f))$ takes the secret key sk, the metadata M, and the file identifier $id(f)$ as input, and outputs a deletion token T_d.

$(T_m, c_m, inf) = \texttt{ModifyToken}(sk, M, id(f), m)$ takes the secret key sk, the metadata M, the file identifier $id(f)$, and the modification m on the file, and creates a modification token T_m with an encrypted modification c_m and information about it inf, to be performed on the file.

$(\{\texttt{accept}, \texttt{reject}\}, M') \leftarrow \texttt{Verify}(sk, M, \mathbf{c}_w, \pi)$ is run on an answer $\mathbf{c}_w$ (empty for updates) and a proof π. It emits an acceptance or a rejection signal, and updates the metadata M (to M') in case of acceptance.

These algorithms will be **executed by the server**:

$(\mathbf{c}_w, \pi) = \texttt{Search}(pk, \mathcal{I}, \mathbf{c}, T_s)$ is an algorithm to find and return the encrypted files containing a keyword. It takes as input the client's public key pk, the encrypted index $\mathcal{I}$, the encrypted file collection $\mathbf{c}$, and the search token T_s, and outputs a subset of the encrypted files $\mathbf{c}_w \subseteq \mathbf{c}$ matching the search token T_s and a proof π showing that the result is authentic, to be sent to the client.

$(\mathbf{c}_w, \pi) = \texttt{BooleanSearch}(pk, \mathcal{I}, \mathbf{c}, T_b)$ is to find and return the encrypted files matching the Boolean combination of the keywords. It takes as input the client's public key pk, the encrypted index $\mathcal{I}$, the ciphertexts $\mathbf{c}$, and the Boolean search token T_b, and outputs a subset of the encrypted files $\mathbf{c}_w \subseteq \mathbf{c}$ and a proof π that shows the result is authentic.

$(\mathcal{I}', \mathbf{c}', \pi) = \texttt{Add}(\mathcal{I}, \mathbf{c}, T_a, c)$ is an algorithm to add a new encrypted file into the collection of encrypted files $\mathbf{c}$ and update the index $\mathcal{I}$ accordingly, given the add token T_a with the new ciphertext c. It outputs the new index $\mathcal{I}'$, the new collection of encrypted files $\mathbf{c}'$, and the proof π showing the operation is performed correctly.

$(\mathcal{I}', \mathbf{c}', \pi) = \texttt{Delete}(\mathcal{I}, \mathbf{c}, T_d)$ given the encrypted index $\mathcal{I}$, the collection of encrypted files $\mathbf{c}$, and the delete token T_d, deletes the specified file and generates the updated index $\mathcal{I}'$ and the file collection $\mathbf{c}'$, along with the proof π.

$(\mathcal{I}', \mathbf{c}', \pi) = \texttt{Modify}(\mathcal{I}, \mathbf{c}, T_m, c_m, inf)$ is run to modify an encrypted file in the collection $\mathbf{c}$ and the encrypted index $\mathcal{I}$ given the modification token T_m, the encrypted modification c_m, and information inf about the modification. It outputs the new index $\mathcal{I}'$, the new (modified) file collection $\mathbf{c}'$, and the proof π.

Definition 2 (Correctness of a VDSSE scheme) *A VDSSE scheme is correct if* $\forall\ k \in \mathbb{N}$, $\forall\ \boldsymbol{f}$, $\forall\ (sk, pk) \leftarrow \texttt{KeyGen}(1^k)$, $\forall\ (\mathcal{I}, \mathbf{c}, M) \leftarrow \texttt{BuildIndex}(sk, \boldsymbol{f})$, *and* $\forall$ *series of update commands, a search query returns only the most recent versions of exactly the file(s) satisfying the criteria, and the file contents are original.*

Definition 3 (Security of a VDSSE scheme) *A VDSSE scheme is secure if it is* ***verifiable and private****, as per definitions below.*

Definition 4 (Verifiability of VDSSE) *We say that a VDSSE scheme is verifiable if no PPT adversary can win the verifiability game with non-negligible probability. The verifiability game below is played between the challenger who acts as the client and the adversary who plays the role of the server.*

Setup The challenger runs $\texttt{KeyGen}(1^k)$ to generate the secret and public keys *(sk,pk)* and shares the public key pk with the adversary. The adversary sends a file collection $\mathbf{f}$ to the challenger who runs $(\mathcal{I}, \mathbf{c}, M) \leftarrow \texttt{BuildIndex}(sk, \mathbf{f})$, and sends the resulting $(\mathcal{I}, \mathbf{c})$ back.

Query The adversary can interact with the challenger polynomially-many times. At each interaction, the adversary *adaptively* asks challenger perform a command $\in$ $(\texttt{Search}, \texttt{BooleanSearch}, \texttt{Add}, \texttt{Delete}, \texttt{Modify})$ of his choice. The challenger then sends back the proper token $T_x \in \{T_s, T_b, T_a, T_d, T_m\}$ with any other necessary information about the update (i.e., c for `Add` or c_m, inf for `Modify`). For each such query, the adversary sends a proof π and any other result (i.e., $\mathbf{c}_w$ for `Search`) to the challenger, who notifies the adversary of the verification result. The challenger applies the verified and accepted changes to her local copy of the file and index storage.

Challenge The challenger sends a query to the adversary, who responds with an answer and proof (e.g., a $\mathbf{c}'_w$ and π' for `Search`). The adversary wins if the answer $\mathbf{c}'_w$ differs from the result of running the query on challenger's local copy, but the proof π' is accepted.

Definition 5 (Privacy of a VDSSE scheme (a.k.a. Dynamic CKA2-security)) *Let* `VDSSE` = (`KeyGen`, `BuildIndex`, `Dec`, `SearchToken`, `Search`, `BooleanSearchToken`, `BooleanSearch`, `AddToken`, `Add`, `DeleteToken`, `Delete`, `ModifyToken`, `Modify`, `Verify`) *be a VDSSE scheme. Consider following experiments with a stateful adversary* $\mathcal{A}$*, a stateful simulator* $\mathcal{S}$*, and stateful leakage functions* $\mathcal{L}_{Init}$*,* $\mathcal{L}_{Srch}$*,* $\mathcal{L}_{BlSrch}$*,* $\mathcal{L}_{AddDel}$*,* $\mathcal{L}_{Mod}$*:*

$\mathbf{Real}_{\mathcal{A}}(k)$: $\mathcal{S}$ generates the keys (sk, pk) by running $\texttt{KeyGen}(1^k)$, and shares the public key pk with $\mathcal{A}$. $\mathcal{A}$ outputs a file collection $\mathbf{f}$. $\mathcal{S}$ generates the encrypted indices and file collection through $(\mathcal{I}, \mathbf{c}, M) \leftarrow \texttt{BuildIndex}(sk, \mathbf{f})$ and sends $(\mathcal{I}, \mathbf{c})$ to $\mathcal{A}$. Then, $\mathcal{A}$ performs a polynomial number of adaptive queries. For each query requested by the adversary, the challenger generates the corresponding token and sends it to $\mathcal{A}$: for a search query, she generates $T_s \leftarrow \texttt{SearchToken}(sk, M, w)$, for a Boolean search query she generates $T_b \leftarrow \texttt{BooleanSearchToken}(sk, M, \phi(w_1, w_2, ..., w_t))$,for an add query she generates $(T_a, c_f) \leftarrow \texttt{AddToken}\ (sk, M, f)$, for a delete query she generates $T_d \leftarrow \texttt{DeleteToken}(sk, M, id(f))$, and for a modify query she generates $(T_m, c_m) \leftarrow \texttt{ModifyToken}(sk, M, id(f), m)$, for the keywords and files of the adversary's choice. Finally, $\mathcal{A}$ outputs a bit b that is the output of the experiment.

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$: $\mathcal{A}$ outputs a file collection $\mathbf{f}$. The simulator $\mathcal{S}$ is *not* given $\mathbf{f}$, but still generates the encrypted index and file collection $(\mathcal{I}, \mathbf{c})$ using the information provided by the leakage $\mathcal{L}_{Init}$, and sends them to $\mathcal{A}$. The adversary makes a polynomial number of adaptive queries. For each query, the simulator is *not* given the information the challenger would have received, but instead is provided by the corresponding leakage $\mathcal{L}_{Srch}$, $\mathcal{L}_{BlSrch}$,$\mathcal{L}_{AddDel}$, or $\mathcal{L}_{Mod}$, using which he prepares and returns an appropriate token along with the required data, e.g., the ciphertext for an add operation, or the content changes for a modification. At the end, $\mathcal{A}$ outputs a bit b that is the output of the experiment.

Our `VDSSE` is ($\mathcal{L}_{Init}$, $\mathcal{L}_{Srch}$, $\mathcal{L}_{BlSrch}$,$\mathcal{L}_{AddDel}$, $\mathcal{L}_{Mod}$)-private against adaptively chosen keyword attacks if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ and a negligible function $\nu(k)$ such that: $|Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1]| \leq \nu(k)$.

4 Construction

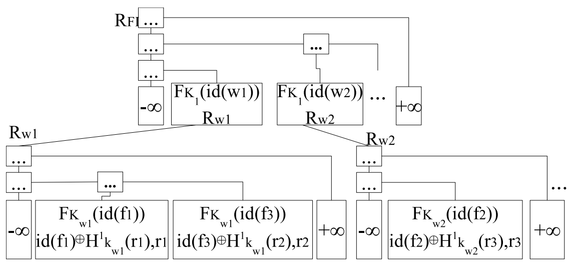
We use a two-level efficient HADS [21] constructed using authenticated skip lists[1] in both levels, to implement the indices. Our scheme consists of three indices: the *forward index*, FI, relating each keyword to the set of file identifiers it appears in, the *inverted index*, II, tying each file to 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.

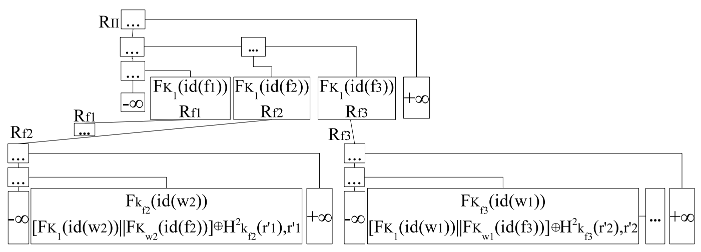
We use these second-level ADSs to prove that this set of files is exactly the set matching the keyword in a search query. We also employ an algorithm operating on these ADSs to generate similar efficient proofs for Boolean search.For a keyword w_i, we associate two keys $K'_{w_i}{=}F_{K_2}(id(w_i))$ 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 4a is shown in Figure 5a.

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.

[1]Similar ADSs, e.g., Merkle hash tree [41] or authenticated 2-3 tree [43] can also be used.



(a) The forward index.



(b) The inverted index.

Figure 5: 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.

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 5b presents a small part of the inverted index construction corresponding to Figure 4b.

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 4.1). 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 [18] or FlexDPDP [19] 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 dictionary $\mathbf{w}$, 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.

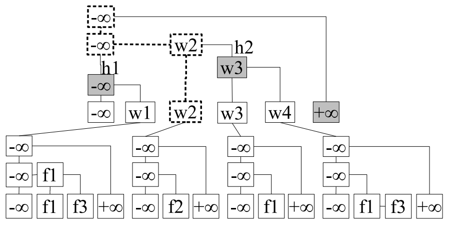
The tokens. Since the indices are encrypted, the client provides the server with the required information about each operation through tokens. The tokens depend on the client's private key, and only the client can generate such tokens. The information required for operations on the file index (according to DPDP) is also sent with each update token. For its details, we refer the reader to [18].

4.1 Search

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 $T_s = (F_{K_1}(id(w_i)), K_{w_i})$.

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 non-membership 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.

[2]This is also similar to the directory-hierarchy extension of DPDP [18], which proves the (non-)existence of the files. Therefore, the client needs to keep only single metadata, regardless of the number of files.


(a) Search.

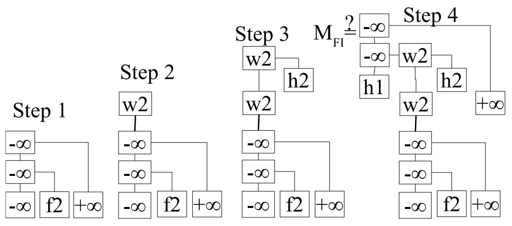
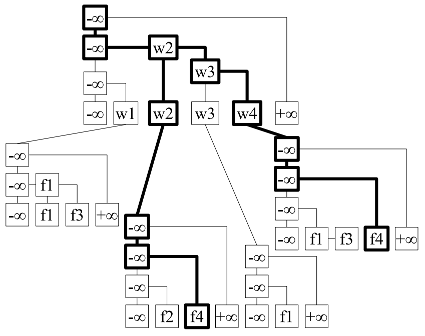
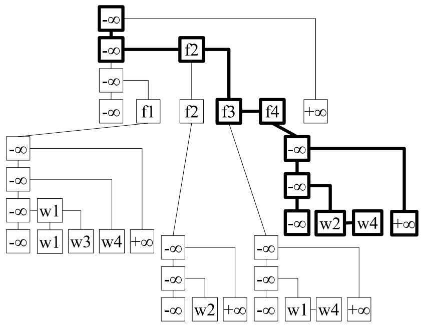

(b) Verification.

Figure 6: Search and verification operations on the forward index.


(a) The forward index.


(b) The inverted index.

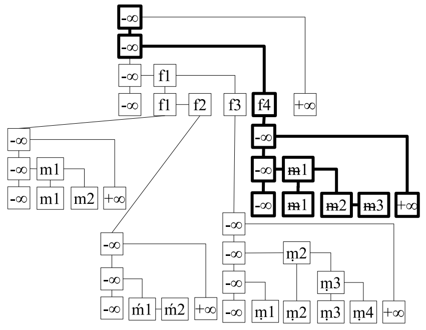

(c) The file index.

Figure 7: The indices after adding a new file f_4 containing w_2 and w_4. The affected parts are bold.

Example. The search token for the keyword w_2 in our example in Figure 4a 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 6a. 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.

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 6b. 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.

4.2 File Addition

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 $T_a{=}(F_{K_1}(id(f_j)), \{key_{w_{i_t}}, val_{w_{i_t}}, F_{K_1}(id(w_{i_t})), key^t_{f_j}, val^t_{f_j}\}_{t=1}^{s})$. 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 4. The token will be $T_a{=}(F_{K_1}(id(f_4)), \{key_{w_2}, val_{w_2}, F_{K_1}(id(w_2)), key^2_{f_4}, val^2_{f_4}, key_{w_4}, val_{w_4}, F_{K_1}(id(w_4)), key^4_{f_4}, val^4_{f_4}\})$. 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 7a. 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 $(key^2_{f_4}, val^2_{f_4})$ and $(key^4_{f_4}, val^4_{f_4})$ 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 7. The server sends proofs generated by each index to the client.

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.

4.3 File Deletion

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 5 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.

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.

4.4 File Modification

Token. The modification is a combination of add and delete tokens, and causes some of the existing relation between the file and its keywords be removed, and some new ones 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.

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.

4.5 Boolean Search

Existing solutions. The existing schemes supporting Boolean search [42, 9, 47, 25] assume a semi-honest adversary: the server executes the query, and finds and sends the resultant files to the client who accepts them without verification. We propose a *dynamic* SSE scheme providing *verifiable* Boolean search. In the malicious setting, the server's answers need to be equipped with a proof including sufficient information enabling the client to check their verifiability. This requires giving more information to the server for the sake of *efficiency*; hence our solution leaks information about individual (encrypted) keywords in the formula. To employ the existing solutions in the malicious setting, for each individual keyword in the search formula, all files containing the keyword should be sent to the client for verification, which is a very inefficient solution [9]. We leave the efficient, verifiable, and privacy-preserving Boolean search in dynamic environments as an open problem.

A **naive solution** in the malicious setting is that the server generates a proof for each keyword in the formula, that shows the set of file identifiers each keyword appears in. Then, he performs a sequence of unions (for '$\vee$'), intersections (for '$\wedge$'), and complementations (for '$\neg$') according to the Boolean search formula, and prepares the final set of file identifiers satisfying the Boolean search criteria. Finally, he sends the resultant files together with the proofs to the client. The client first verifies each proof, and if all were accepted, performs a sequence of unions, intersections, and complementations on the accepted proof results, similar to the server. Moreover, she checks the integrity of all received files. At the end, she accepts the answer if the set of file identifiers that are the output of the verification, exactly matches the received files. We propose a much more efficient approach below.

Token. For each keyword in the Boolean formula, the server needs the same information as a single-keyword search. Therefore, for a Boolean combination of t keywords, $\phi(w_1, ..., w_t)$, we create one search token T_s per w_i to form the Boolean search token.

Server computation. In the malicious setting, the server can perform better, especially when some involved second-level ADSs are very large, where only a very small part satisfies the query. Therefore, by operating on the keyword proofs, he can generate efficient proofs.

We decompose a Boolean search formula to a set of *primitive combinations* for which a proof using the existing ADSs can be generated. Each proof provides an authenticated set of file identifiers to the client, used for generating the final authenticated set of file identifiers by applying a sequence of unions, intersections, and complementations, according to the Boolean formula. Our key contribution here is that these primitive combinations are not just for a single keyword, as in the naive approach. Any Boolean combination of keywords can be constructed using these **primitive combinations**:

- $\neg\varphi(w_1, ..., w_t)$: The server sends the proof for $\varphi(w_1, ..., w_t)$ to the client who will consider the complement file set to be the query answer if the proof is accepted.
- $w_1 \vee w_2 \vee ... \vee w_t$: The proof should include at least all *distinct* file identifiers stored at each ADS corresponding to a w_i. The server can traverse all these ADSs once and generate a space-efficient proof. The client uses the proof to reconstruct the ADSs for verification. Alternatively, the server sends proofs of all w_is (i.e., the file identifiers in the corresponding ADSs together with the information required for reconstructing them) to the client, without further computation. The client reconstructs the ADSs one-by-one, and if all are accepted, outputs the set of distinct file identifiers as the authentic result. This is a trade-off between the communication and client and server computations.
- $\diamond_1 w_1 \wedge \diamond_2 w_2 \wedge ... \wedge \diamond_t w_t$ where $\diamond_i$ is either empty or $\neg$: If all $\diamond_i$s are $\neg$ (i.e., the formula is $\neg w_1 \wedge \neg w_2 \wedge ... \wedge \neg w_t$), it can be converted into $\neg(w_1 \vee w_2 \vee ... \vee w_t)$, using De Morgan's laws. Otherwise, we have a set of keywords, each with a related second-level ADS, connected by '$\wedge$'. If the file identifiers were stored in these second-level ADSs *sorted* according to their plain values, we could easily compare them and find the common values. But they are stored sorted on their encrypted values, hence, their plain values cannot be efficiently compared. Therefore, we first sort these second-level ADSs according to their sizes (since the order of keywords is not important), and start by the comparing the two smallest ones. For each file identifier in the first ADS, we generate either a membership or a non-membership proof in the second ADS, depending on whether it appears in the second ADS or not. This gives us two sets of file identifiers: one set including those that appear in the second ADS and the other containing those that do not. We continue comparing the former set with the next ADS similarly, while leaving out the latter one. This process goes on until all ADSs are accomplished. The important idea behind this process is that the set of file identifiers to compare with the next ADS gets smaller as we go ahead. The algorithm is illustrated in F.

Example. Consider the query $Q{=}w_2 \vee \neg(w_1 \wedge w_3)$ to be executed on Figure 4. The client prepares the token $T_b{=}(F_{K_1}(id(w_2)), K_{w_2}) \vee \neg((F_{K_1}(id(w_1)), K_{w_1}) \wedge (F_{K_1}(id(w_3)), K_{w_3}))$. The server decomposes it into primitives $Q_1{=}(F_{K_1}(id(w_2)), K_{w_2})$ and $Q_2{=}((F_{K_1}(id(w_1)),\ K_{w_1}) \wedge (F_{K_1}(id(w_3)), K_{w_3}))$, and generates their proofs separately using the related second-level ADSs. The proof of Q_1 is an ADS proof as $\pi_{Q_1}=\{h(-\infty), id(f_2),\ h(+\infty)\}$. For Q_2, we use the algorithm 1 that generates $\pi_{Q_2}= \{h(-\infty); \{id(f_1){:}h(-\infty), id(f_1), h(id(f_3)),\ h(+\infty)\}; h(+\infty)\}$, which obeys the format: $\{h(-\infty); \{id(f_i){:}\pi_{id(f_i)}\ \mathit{using}\ ADS_{w_3}\}_{id(f_i)\in\ ADS_{w_1}}; h(+\infty)\}$. It includes the file identifiers in ADS_{w_1}, each followed by a membership proof in ADS_{w_3}. The *authentic* result of Q_1 is $\{id(f_2)\}$, and that of Q_2 is $\{id(f_1)\}$.

Then, the server performs a complementation on the authentic result of Q_2, which yields $\{id(f_2), id(f_3)\}$, followed by a union with the authentic result of Q_1, achieving the final result $\{id(f_2), id(f_3)\}$. Moreover, he generates the integrity proof, π_{int}, for the files in the set $\{id(f_2), id(f_3)\}$ using the file index. Finally, the server sends the proofs π_{Q_1}, π_{Q_2}, and π_{int}, together with the encrypted files f_2 and f_3 to the client.

The client first verifies π_{Q_1} and π_{Q_2}. If they were accepted, she performs the set operations as the server, and finds $\{id(f_2), id(f_3)\}$. Then, she checks that there must be *only* two files f_2 and f_3 in the answer. Finally, she accepts the answer if π_{int} is also verified.

4.6 VDSSE in the Standard Model

Our scheme employs one-time pad encryption using two hash functions modeled as random oracles. This is necessary in our proof so that the simulator can later claim he encrypted the correct value, even though during the encryption process he had no idea about it.Alternatively, one can use a deniable encryption scheme [6, 17] or a non-committing encryption scheme [14]. It is claimed that [52] and [8] 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 non-committing encryption scheme: For any ciphertext, a key matching the ciphertext to the desired message can easily be found. It can effectively replace our random oracle use.

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.

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 token generation and the server computation are exactly as in the ROM. We substitute each keyword in a Boolean search formula with a corresponding token T_s to generate the Boolean search token.

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 token generation and the server computation are exactly as in the ROM.

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 the node is empty. This means that deletion will not reduce the token size of later searches.

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 token generation and server computation are exactly as in the ROM.

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.

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. But, 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 [8] 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.

5 Analysis

5.1 Security

Before going into details of the security proof, we formalize our leakage functions:

$\mathcal{L}_{Init}$ shows the information leakage during the initialization: $\mathcal{L}_{Init}(\mathbf{f})$=($|\mathbf{f}|$, $|\mathbf{f}_{w_i}|_{w_i\in\mathbf{w}}$, $|\mathbf{w}|$, $|\mathbf{w}_{f_j}|_{f_j\in\mathbf{f}}$, $|f_j|_{j=1}^{n}$, $\{eid(f_j)\}_{f_j\in\mathbf{f}}$, $\{eid(w_i)\}_{w_i\in\mathbf{w}}$). Briefly, the number of files and keywords, the number of keywords per file and files per keyword leak, as well as the file sizes and encrypted keywords ($eid(w_i)$s) and file identifiers ($eid(f_j)$s). The keywords and the files themselves never leak, as they are kept encrypted.

$\mathcal{L}_{Srch}$ shows the encrypted file identifiers containing $eid(w_i)$ (the access pattern of w_i) revealed during the search operation: $\mathcal{L}_{Srch}(\mathbf{f},w_i) = (eid(w_i), \{eid(f_j)\}_{w_i\in f_j})$.

$\mathcal{L}_{BlSrch}$ is similar to $\mathcal{L}_{Srch}$ but for a Boolean combination of keywords: $\mathcal{L}_{BlSrch}(\mathbf{f},\mathbf{w}'\subseteq\mathbf{w}) = \{eid(w_i), \{eid(f_j)\}_{w_i\in f_j}\}_{w_i\in\mathbf{w}'}$. The leakage is the set of encrypted keyword identifiers, $eid(w_i)$, each with its access pattern.

$\mathcal{L}_{AddDel}$ shows the leakage during a file addition or deletion: its size, and the encrypted identifier of the file and its keywords. $\mathcal{L}_{AddDel}(\mathbf{f},f_j)$=$(eid(f_j), |f_j|, \{eid(w_i)\}_{w_i\in f_j})$.

$\mathcal{L}_{Mod}$ contains the leakage during a modification: the random values used as the inputs of the hash function for hiding the information about the deleted keywords (leaked in the first round), size of the file after modification, and the set of encrypted keyword identifiers being added (of size t_1) and deleted (of size t_2), leaked in the second round. Finally, $\mathcal{L}_{Mod}(\mathbf{f},f_j)$=$(eid(f_j), |f_j^{new}|, \{eid(w_{i_t})\}_{t=1}^{t_1}, \{eid(w_{i'_t}), r_{i'_t}\}_{t=1}^{t_2})$. In the standard model, the modify leakage contains the location of the deleted and added keywords: $\mathcal{L}_{Mod}(\mathbf{f},f_j)$=$(eid(f_j), |f_j^{new}|, \{l_t, eid(w_{i_t})\}_{t=1}^{t_1}, \{l'_t, eid(w_{i'_t})\}_{t=1}^{t_2})$.

Theorem 1 *If a CPA-secure symmetric-key encryption scheme* SKE*, a secure HADS scheme [21], a secure PRF F, and two hash functions H^1 and H^2 modeled as random oracles are employed, then our VDSSE construction is secure according to Definition 3.*

Theorem 2 *If a CPA-secure symmetric-key encryption scheme* SKE*, a secure HADS scheme, and secure PRFs F, G are employed, our VDSSE scheme in the standard model is secure according to Definition 3.*

We prove these formally by proving verifiability and privacy, in A and B.

Table 2: Table for comparison of dynamic SSE schemes. (n=$|\mathbf{f}|$ is the number of files, m=$|\mathbf{w}|$ is the number of keywords, $d = |\mathbf{f}_w|$ is the number of files containing w, N is the number of occurrences of all keywords in all files, and B is the size of a Bloom filter. 'T. Size' stands for 'Token Size'. $VDSSE_{RO}$ and $VDSSE_{ST}$ are our schemes in random oracle and standard model, respectively.)

Scheme	**Storage**		**Search**		**Add**		**Delete**	
	Client	**Server**	**T. size**	**Computation**	**T. size**	**Computation**	**T. size**	**Computation**
[55]	$O(m)$	$O(N)$	$O(1)$	$O(d)$	$O(w)$	$O(w)$	$O(m)$	$O(N)$
DSSE [35]	$O(1)$	$O(N)$	$O(1)$	$O(d)$	$O(w)$	$O(w)$	$O(1)$	$O(wd)$
PDSSE [34]	$O(1)$	$O(nm)$	$O(1)$	$O(d \log n)$	$O(w \log n)$	$O(w \log n)$	$O(w \log n)$	$O(w \log n)$
PDSE [52]	$O(\log N)$	$O(N)$	$O(\log N)$	$O(d \log^3 N)$	$O(w)$	$O(w \log^2 N)$	$O(w)$	$O(w \log^2 N)$
BS [47]	$O(1)$	$O(nB)$	$O(\log n)$	$O(bd \log n)$	$O(B)$	$O(B)$	$O(1)$	$O(1)$
[8]	$O(1)$	$O(N)$	$O(1)$	$O(d)$	$O(w)$	$O(w)$	$O(w)$	$O(w)$
MCBS [25]	$O(1)$	$O(nB)$	$O(\log n)$	$O(bd \log n)$	$O(B)$	$O(B)$	$O(1)$	$O(1)$
$VDSSE_{RO}$	$O(1)$	$O(N)$	$O(1)$	$O(\log m + d)$	$O(w)$	$O(\log n + w \log(md))$	$O(1)$	$O(\log n + w \log(md))$
$VDSSE_{ST}$	$O(n+m)$	$O(N)$	$O(d)$	$O(\log m + d)$	$O(w)$	$O(\log n + w \log(md))$	$O(w)$	$O(\log n + w \log(md))$

5.2 Comparison to Previous Work (Asymptotic)

Table 2 presents an asymptotic efficiency comparison among recent dynamic SSE schemes.

Storage. The client storage in our scheme is optimal ($O(1)$). Server storage in all schemes are comparable. The client storage in our standard model solution is $O(n + m)$. This is very small compared to the size of all outsourced files, since for each (large) file, only a few hundred bits are stored. However, this amount of storage may be noticeable when the number of keywords or files is very large. The same holds for the standard model extension of PDSE and [8].

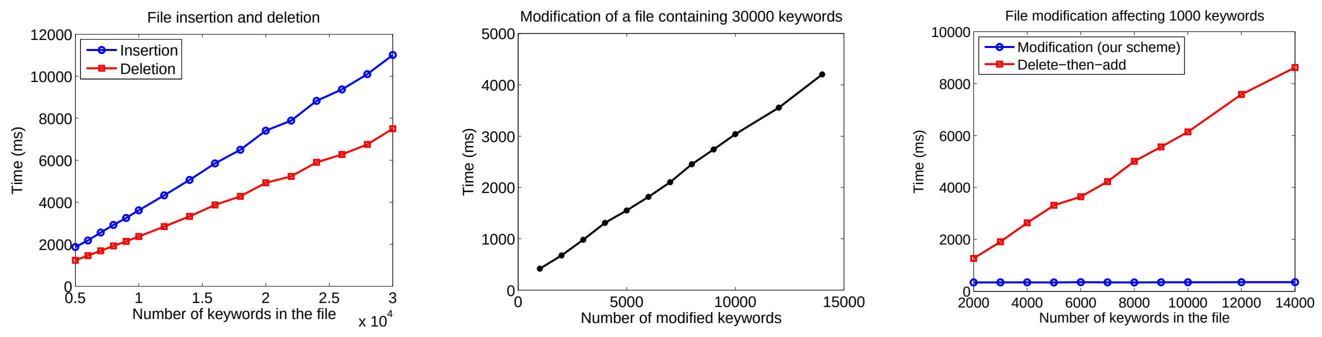
Token size. Our VDSSE, together with DSSE, BS, and MCBS have optimal token sizes (constant-size search and delete, $O(w)$ add tokens). In the standard model, we need to compute and send the PRF outputs for all leaf nodes of the related second-level ADS, and hence the token sizes increase. Therefore, the search token is $O(d)$, and the add and delete tokens are $O(w)$. The token sizes of the standard model extension of PDSE are, however, $O(\log N)$ times worse than those of our scheme.

Computation. DSSE possesses the best search time $O(d)$, add time $O(w)$, and delete time $O(wd)$. Our scheme naturally requires more computation for supporting verifiability and file integrity. We pay an *additive* (not multiplicative) $O(\log m)$ cost for a membership proof of a keyword, and an *additive* $O(\log n)$ cost for a membership proof of a file. Removing the verifiability and integrity proofs causes our scheme's complexities drop to those of DSSE. In our standard model construction, the client computation increases to $O(d)$ for generating the search token, and to $O(w)$ for generating the delete tokens, but the server computation is the same as in the random oracle model. The standard model client and server computation times of PDSE are again $O(\log N)$ times worse than those of our scheme.

To sum up, our schemes are **dynamic** SSE schemes supporting **modifiability** and efficient **Boolean search** in the **malicious** setting. Furthermore, our standard model construction is the most efficient such construction, supporting verifiable dynamic operations and Boolean search, with full security proof. Following, we provide concrete performance numbers and demonstrate the practical efficiency of our system.

5.3 Performance Analysis

Setup. To evaluate our SSE scheme, we implemented a prototype with the two-level efficient HADS construction [21] with Flexlist [20] 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.



(a) File addition and deletion. (b) File modification. (c) File modification.

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

5.3.1 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.

File addition and deletion. Figure 8a 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 seconds, respectively.

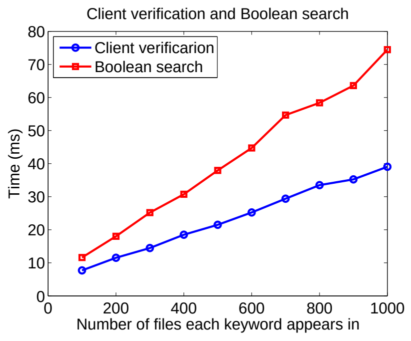
File modification. Figure 8b depicts the results of a file modification affecting a different number of keywords when the file already contains 30,000 keywords. A modification affects a set of keywords on the indices, and hence, the operation time increases with the number of affected keywords. But for 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 8c; 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 8c 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.

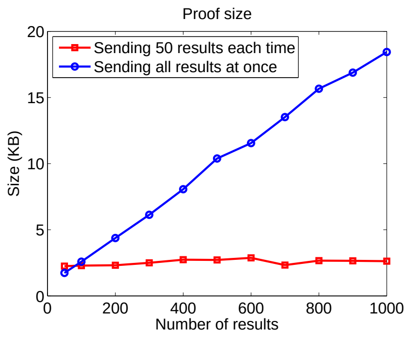
File search. The server uses the first-level ADS of the forward index to generate the membership proof. Then, he decrypts all the encrypted file identifiers stored at the leaf nodes of the corresponding second-level ADS, and sends all the files along with the proof to the client. Hence, the search time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

Boolean search. We performed the Boolean search of the form $w_1 \wedge w_2$ for different number of files (sharing w_1 and w_2), assuming both keywords appear in almost the same number of files (of which a portion contains both keywords together). The server needs to traverse the second-level ADSs of all keywords in the formula, and generate their membership proof at the first level of the forward index. The client, however, needs to reconstruct the whole second-level ADSs and the proof paths on the first-level ADS for verification. This is why the client verification times are greater than the server computation times in Figure 9a. Our Boolean search at the server takes 8 ms when each keyword appears in 100 files and 40 ms when each keyword appears in 1000 files. However, it takes about 11 and 74 ms for the client to verify them.

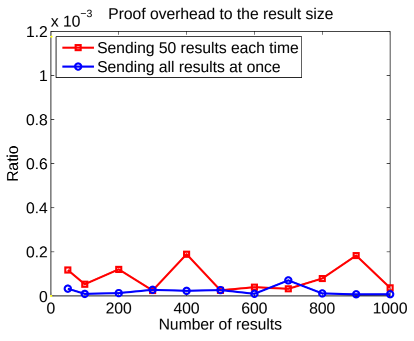
Iterated search. The server can send all the search results together, or may send them in small groups



(a) Boolean search.

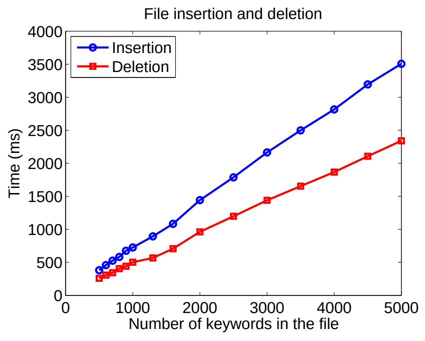


(b) Proof size.

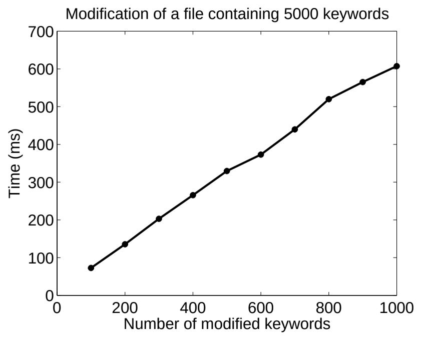


(c) Proof overhead.

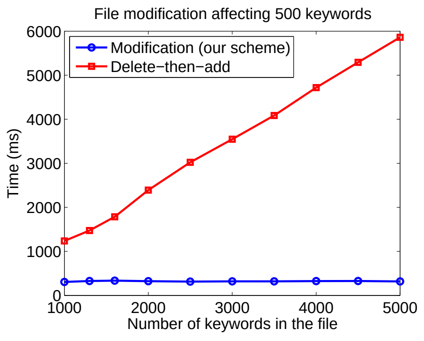
Figure 9: Proof generation and verification, and proof size and overhead to the query result.



(a) File addition and deletion.



(b) File modification.



(c) File modification.

Figure 10: File addition, deletion, and modification for the second scenario.

(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 9b 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.

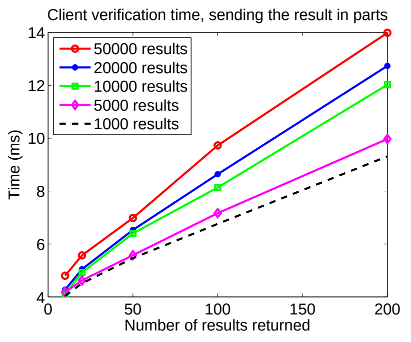
Proof overhead of our scheme is very insignificant compared to the size of resultant files being transferred, e.g., about 0.01% of the search result size as shown in Figure 9c, and can be neglected.

5.3.2 Second Scenario: Large Number of Small Documents

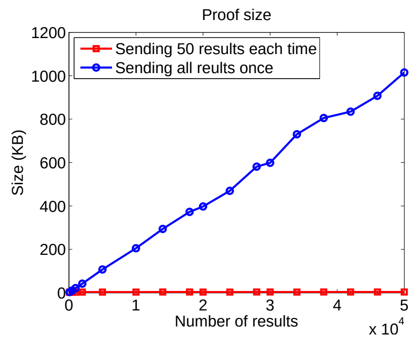
This scenario corresponds to the case that a large number (e.g., 100,000) of webpages are outsourced to an untrusted cloud server. We investigated the number of distinct keywords in different websites (e.g., bbc.com, office.com, nytimes.com, and other websites related to news, technology, and education) using the online word counter tool from words.contentor.com and realized that the number of distinct words in a typical webpage is generally between 100 and 5000.

File addition and deletion. The respective times for this scenario are depicted in Figure 10a. It shows a reduction compared to Figure 8a since each file contains smaller number of distinct keywords (i.e., at most 5000). Figure 10a reveals that the addition of a new file including 500 and 5000 distinct keywords takes about 420 and 3500 ms, respectively.

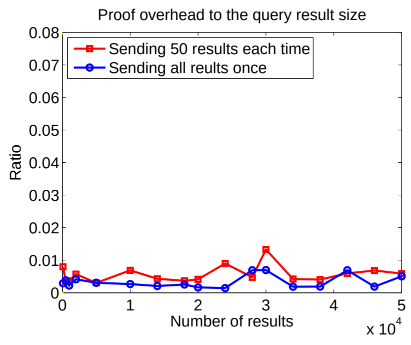
[3]If the search results of all keywords are stored ranked, the client is normally interested in those with high ranks, and may not want to receive those with low ranks [56, 46].



(a) Client verification time.



(b) Proof size.



(c) Proof overhead.

Figure 11: The proof size and proof overhead compared to the query result.

File modification. In a similar manner, the file modification shows a drop compared to the previous scenario, as shown in Figure 10b. It ranges from 80 to 600 *ms* when the number of keywords increases from 100 to 1000. Once again, Figure 10c 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.

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*.

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 [21] 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. But 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 11a, and are around 4 to 14 *ms*.

Proof overhead. Our scheme generates very small proofs, as illustrated in Figure 11b. 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 11c.

5.3.3 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.

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.

Table 3: Table for comparison of search times (ms) in our scheme and [8].

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

Second scenario. For 100,000 files (n=100,000) and 100,000 keywords (m=100,000), the client storage will be 200,000×20=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 10,000×128=1,280,000 bits$\simeq$156 KB when a keyword matches 10,000 files. The add token size is 128 + 5000×(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.

5.3.4 Comparison to Previous Work (Concrete)

Cash *et al.* [8] 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 3. 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 more than an order of magnitude worse, compared to our solution. This is due to the fact that in Cash *et al.* [8] work, 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.* [60] generalized the work of Cash *et al.* [8] causing time overhead on the order of microseconds. Thus, our time comparison with Cash *et al.* [8] immediately applies to the comparison against the Zhu *et al.* [60] scheme.

6 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. Our VDSSE supports efficient verifiable Boolean search in general, not only conjunction, though leaking information about individual keywords in the Boolean formula. Using our approach, the server generates a space-efficient proof for the whole Boolean search result, to be verified by the client. We also presented a dynamic construction secure in the **standard model** with full security proof via simulation, Boolean search capability, and performance evaluation for the first time. 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 $\simeq$ 362 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] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS'07; Alexandria, VA, USA*, pages 598–609. ACM, 2007.

[2] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm'08; İstanbul, Turkey*, pages 9:1–9:10. ACM, 2008.

[3] C. Bosch, A. Peter, B. Leenders, H. W. Lim, Q. Tang, H. Wang, P. Hartel, and W. Jonker. Distributed searchable symmetric encryption. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 330–337. IEEE, 2014.

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

[5] K. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *CCS'09*, pages 187–198. ACM, 2009.

[6] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *CRYPTO'97; Santa Barbara, CA, USA*, pages 90–104. Springer, 1997.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS'15*, CCS '15, pages 668–679, New York, NY, USA, 2015. ACM.

[8] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS'14; San Diego, CA, USA*, pages 23–26, 2014.

[9] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO'13; Santa Barbara, CA, USA*, pages 353–373. Springer, 2013.

[10] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT'13; Athens, Greece*, pages 279–295. Springer, 2013.

[11] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. In *TCC*, 2014.

[12] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS'05; New York, NY, USA*, pages 442–455. Springer, 2005.

[13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT'10; Singapore*, pages 577–594. Springer, 2010.

[14] S. G. Choi, D. Dachman-Soled, T. Malkin, and H. Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *ASIACRYPT'09; Tokyo, Japan*, pages 287–302. Springer, 2009.

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

[16] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS'08*, pages 411–420. IEEE, 2008.

[17] M. Dürmuth and D. M. Freeman. Deniable encryption with negligible detection probability: An interactive construction. In *EUROCRYPT'11*. Springer, 2011.

[18] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS'09; Alexandria, VA, USA*, pages 213–222. ACM, 2009.

[19] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and Ö. Özkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. *ACM Transactions on Storage*, 12(4):23:1–23:44, 2016.

[20] E. Esiner, A. Küpçü, and O. Özkasap. Analysis and optimizations on flexdpdp: A practical solution for dynamic provable data possession. In *ICC'14; Muscat, Oman*, pages 65–83. Springer, 2014.

[21] M. Etemad and A. Küpçü. Database outsourcing with hierarchical authenticated data structures. In *ICISC'13; Seoul, Korea*, pages 381–399. Springer, 2013.

[22] M. Etemad and A. Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *ACNS'13; Banff, Alberta, Canada*, pages 1–18. Springer, 2013.

[23] M. Etemad and A. Küpçü. Verifiable database outsourcing supporting join. *Journal of Network and*

Computer Applications, 115:1–19, 2018.

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

[25] B. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: A scalable private dbms. In *IEEE Symposium on Security and Privacy (SP'15); San Jose, CA, USA*, pages 395–410. IEEE, 2015.

[26] S. Garg, P. Mohassel, and C. Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual Cryptology Conference*, pages 563–592. Springer, 2016.

[27] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

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

[29] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Automata, Languages and Programming*, pages 576–587. Springer, 2011.

[30] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[31] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS'12*, 2012.

[32] A. Juels and B. S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *ACM CCS'07; Alexandria, VA, USA*, pages 584–597. ACM, 2007.

[33] S. Kamara and K. Lauter. Cryptographic cloud storage. *Financial Cryptography and Data Security*, pages 136–149, 2010.

[34] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. *Financial Cryptography and Data Security (FC'13); Okinawa, Japan*, pages 258–274, 2013.

[35] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM CCS'12; Alexandria, VA, USA*, pages 965–976. ACM, 2012.

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

[37] A. Küpçü. Official arbitration with secure cloud storage application. *The Computer Journal*, 58(4):831–852, 2015.

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

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

[40] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[41] R. C. Merkle. A certified digital signature. In *CRYPTO'89; Santa Barbara, CA, USA*, pages 218–238. Springer, 1989.

[42] T. Moataz and A. Shikfa. Boolean symmetric searchable encryption. In *ACM CCS'13; Alexandria, VA, USA*, pages 265–276. ACM, 2013.

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

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

[45] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Security and Privacy (SP'14); San Jose, CA, USA*, pages 639–654. IEEE, 2014.

[46] C. Örencik and E. Savaş. An efficient privacy-preserving multi-keyword search over encrypted cloud

data with ranking. *Distributed and Parallel Databases*, 32(1):119–160, 2014.

[47] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *IEEE Symposium on Security and Privacy (SP'14); San Jose, CA, USA*, pages 359–374. IEEE, 2014.

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

[49] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT'08; Melbourne, Australia*, pages 90–107. Springer, 2008.

[50] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336. ACM, 2013.

[51] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*. IEEE, 2000.

[52] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS'14; San Diego, CA, USA*, pages 72–75, 2014.

[53] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *Parallel and Distributed Systems, IEEE Transactions on*, 25(11):3025–3035, 2014.

[54] R. Tamassia. Authenticated data structures. In *ESA'03; Budapest, Hungary*, pages 2–5. Springer, 2003.

[55] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *SDM'10; Singapore*, pages 87–100. Springer, 2010.

[56] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *ICDCS'10; Genova, Italy*, pages 253–262. IEEE, 2010.

[57] J. Wang, H. Ma, Q. Tang, J. Li, H. Zhu, S. Ma, and X. Chen. Efficient verifiable fuzzy keyword search over encrypted data in cloud computing. *Computer Science and Information Systems*, 10(2):667–684, 2013.

[58] M. Yoshino, K. Naganuma, and H. Satoh. Symmetric searchable encryption for database applications. In *NBiS'11*, pages 657–662. IEEE, 2011.

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

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

A Security Proof in the Random Oracle Model

Theorem 3 *Our VDSSE scheme is verifiable according to Definition 4, provided that the underlying HADS [21] scheme is secure.*

Proof 1 *We reduce verifiability of our VDSSE scheme to the security of the underlying building blocks, the three HADSs: FI, II, FX. If a PPT adversary $\mathcal{A}$ wins the VDSSE verifiability game with non-negligible probability, we use it to construct a PPT algorithm $\mathcal{B}$ who breaks security of at least one of the HADSs, with non-negligible probability. $\mathcal{B}$ acts as the server in the HADS security game played with the HADS challengers $\mathcal{C}_{FI}$, $\mathcal{C}_{II}$, and $\mathcal{C}_{FX}$. Simultaneously, he plays the role of the challenger in the VDSSE verifiability game with $\mathcal{A}$. He receives public keys of the HADSs from their challengers and relays them to $\mathcal{A}$.*

During the setup phase, $\mathcal{B}$ receives commands from $\mathcal{A}$, and forwards each part to the corresponding HADS challenger in its respective format. At the same time, $\mathcal{B}$ builds a local copy of the HADS structures for herself that is invisible to the adversary $\mathcal{A}$, and thus will not affect his behavior. After the setup phase, $\mathcal{A}$ selects a command, generates the answer and proof for the command, and sends them to $\mathcal{B}$. The adversary wins, *if the answer is different from the real answer ($\mathcal{B}$ can find the real answer since he maintains a local copy), and the proof is verified.*

The proof has three parts: $\pi = \pi_{FI}||\pi_{II}||\pi_{FX}$. $\mathcal{B}$ forwards the command, answer, and proof parts to the corresponding challengers. If $\mathcal{A}$ passes the VDSSE verification with non-negligible probability p, $\mathcal{B}$ can also pass each of the HADS verifications with non-negligible probability p (breaking the HADS security).

Since the HADS is secure, p must be negligible, which means that $\mathcal{A}$ can break verifiability of the VDSSE only with negligible probability. Therefore, if the underlying HADS schemes are secure, our VDSSE scheme is verifiable.

Theorem 4 *If SKE is a CPA-secure symmetric-key encryption scheme, the HADS is secure, F is a secure PRF, and the hash functions H^1 and H^2 are modeled as random oracles, then our VDSSE scheme is $(\mathcal{L}_{Init}, \mathcal{L}_{Srch}, \mathcal{L}_{BlSrch}, \mathcal{L}_{AddDel}, \mathcal{L}_{Mod})$-private against adaptive chosen-keyword attacks.*

Proof 2 ***Overview:*** *We construct a PPT simulator $\mathcal{S}$ who simulates the client in a way that is indistinguishable from the real client by any PPT distinguisher. The simulator starts by constructing a simulated (encrypted) index $\mathcal{I}$ and a simulated version of the collection of encrypted files $\mathbf{c}$. These simulated versions are generated as randomly selected values, but the simulator uses the information provided by $\mathcal{L}_{Init}$ to make them similar to the real ones. The simulator needs the number of files, their sizes, the number of keywords in each file, the number of keywords in the dictionary, and the number of files per keyword for a correct simulation of the initial index. These are all given to the simulator in $\mathcal{L}_{Init}$. Since the values stored at the forward and inverted indices are the outputs of PRFs, the simulator can select random values instead. The PRF security guarantees that no PPT distinguisher can distinguish them. $\mathcal{S}$ uses the encryption of all-zero strings of given sizes for the files; an action guaranteed to be indistinguishable by the CPA-security of the encryption scheme used. The tokens are also simulated in a similar manner, except that $\mathcal{S}$ should keep a local copy to keep the tokens consistent; the problem that is pointed out and solved first in [15], and later in [13, 35]. She performs all simulation computations on her own local copy. Now, we detail the construction of such a PPT simulator $\mathcal{S}$ who adaptively simulates the encrypted files (ciphertexts), the indices, and tokens. We refer to the encrypted keyword and file identifiers (i.e., $F_{K_1}(id(w_i))$ and $F_{K_1}(id(f_j))$) stored at the first-level ADSs and are visible to the adversary, as eid(.). However, $id(f_j)$ refers to name under which the file is stored at the server.*

File index *We discuss only the forward and inverted indices below. The first-level ADS of the file index is the same as that of the inverted index; hence, they are simulated similarly. The second-level ADSs of the file index are DPDP constructions that can be built like an honest client over the simulated encrypted files.*

Initialization *The simulator, after generating K_5 via KeyGen, builds some special internal data structures required for performing simulation. FS is a vector of size n used for storing the random hash function keys assigned to each file identifier (to be used at the related second-level ADS of the inverted index). A similar vector of size m, WS, stores the random hash function keys assigned to the keywords. For each keyword w_i, a matrix WS_{w_i} of size $\boldsymbol{f}_{w_i} \times 3$ is assigned to store the random keys and values and randomnesses that are generated to build the FI_{w_i}. Similarly, for each file f_j, a matrix FS_{f_j} of size $\boldsymbol{w}_{f_j} \times 3$ is assigned to store the random keys and values and randomnesses required for constructing the II_{f_j}. Moreover, two matrices WO and FO, that are empty at the outset, are used to answer the random oracle queries for H^1 and H^2, respectively.*

For each encrypted keyword and file identifier eid(.) given by $\mathcal{L}_{Init}$, $\mathcal{S}$ generates random k-bit hash function keys k_i and k'_j and registers them in WS and FS, respectively (i.e., $WS[eid(w_i)] = k_i$ and $FS[eid(f_j)] = k'_j$). Moreover, for each keyword w_i, $|\boldsymbol{f}_{w_i}|$-many l-bit random values key_{f_j} and val_{f_j} (replacing $(id(f_j) \oplus H^1_{K_{w_i}}(r_j))$) and k-bit random values r_{f_j} are generated and stored in WS_{w_i}; i.e., $\{WS_{w_i}[t][1] = key_{f_j}; WS_{w_i}[t][2] = val_{f_j}; WS_{w_i}[t][3] = r_{f_j}\}_{t=1}^{|\boldsymbol{f}_{w_i}|}$. Similarly, for each file identifier f_j, $|\boldsymbol{w}_{f_j}|$-many l-bit random values key_{w_i}, 2l-bit random values val_{w_i} (replacing $[F_{K_1}(id(w_i))||key_{f_j}] \oplus H^2_{K_{f_j}}(r_i)$) and k-bit random values r_{w_i} are generated and stored in FS_{f_j}; i.e., $\{FS_{f_j}[t][1] = key_{w_i}; FS_{f_j}[t][2] = val_{w_i}; FS_{f_j}[t][3] = r_{w_i}\}_{t=1}^{|\boldsymbol{w}_{f_j}|}$. These values are indistinguishable for any PPT distinguisher, since the outputs of H^1 and

H^2 are random. To simulate each encrypted file $f_j \in \boldsymbol{f}$, she encrypts an all-zero string of length $|f_j|$ as $c_j \leftarrow \texttt{EncSKE}_{K_5}(0^{|f_j|})$).

*Now, she performs **BuildIndex** using these random values instead of the actual ones. Finally, she programs the random oracles to make these values consistent with future queries: $\{WO[WS[eid(w_i)]][WS_{w_i}[t][3]] = WS_{w_i}[t][2] \oplus id(f_{j_t})\}_{t=1}^{|\boldsymbol{f}_{w_i}|}$ and $FO[FS[eid(f_j)]][FS_{f_j}[t][3]] = FS_{f_j}[t][2] \oplus \langle eid(w_{i_t})||WS_{w_{i_t}}[l_t][1]\rangle\}_{t=1}^{|\boldsymbol{w}_{f_j}|}$. Consistency means that the simulator generates search tokens in a way that the adversary observes the same set of file identifiers for the same keyword. If any add, modify, and delete operation occurs in between, this is reflected correctly in the results the adversary observes.*

Simulating a single-keyword search token *Using the information in the leakage function $\mathcal{L}_{Srch}(\boldsymbol{f}, id(w_i)) = (eid(w_i), \{eid(f_j)\}_{w_i \in f_j})$, the simulator finds the random value and key assigned to $eid(w_i)$, and outputs the token as $T_s = (eid(w_i), WS[eid(w_i)])$, and programs WO to reflect the file identifiers in the leakage as $\{WO[WS[eid(w_i)]][WS_{w_i}[l_{f_j}][3]] = WS_{w_i}[l_{f_j}][2] \oplus eid(f_j)\}_{eid(f_j)\in\mathcal{L}_{Srch}}$, where l_{f_j} is the location of $eid(f_j)$ in WS_{w_i}.*

Simulating a Boolean search token *The simulator takes the same steps as for one-keyword case for each keyword in $\mathcal{L}_{BlSrch}$.*

Simulating the add token *The simulator updates her local data structures according to the information in $\mathcal{L}_{AddDel}(\boldsymbol{f}, f_j) = (eid(f_j), |f_j|, \{eid(w_i)\}_{w_i \in f_j})$, in the same way as in initialization: She generates a k-bit random hash function key for this file, registers it in FS, builds a second-level ADS containing all keyword identifiers, and ties its root to the inverted index with the key $eid(f_j)$. She also adds $eid(f_j)$ into the second-level ADS of all keywords in $\mathcal{L}_{AddDel}$. Finally, she outputs $T_a = (eid(f_j), \{FS_{f_j}[t][1], FS_{f_j}[t][2]||FS_{f_j}[t][3], eid(w_{i_t}), WS_{w_{i_t}}[l_t][1], WS_{w_{i_t}}[l_t][2]||WS_{w_{i_t}}[l_t][3]\}_{eid(w_{i_t})\in\mathcal{L}_{AddDel}})$, where t is the location of $eid(w_{i_t})$ in FS_{f_j} and l_t is the location of $eid(f_j)$ in $WS_{w_{i_t}}$.*

Simulating the delete token *$\mathcal{S}$ uses the information in $\mathcal{L}_{AddDel}(\boldsymbol{f}, f_j) = (eid(f_j), |f_j|, \{eid(w_i)\}_{w_i \in f_j})$ to check through FS if a hash function key is already assigned to $eid(f_j)$. If not, she first generates a new k-bit random key k'_j and sets $FS[eid(f_j)] = k'_j$. Then, she programs FO in a way that assigns the file identifiers given by $\mathcal{L}_{AddDel}$ to the random values in FS_{f_j}. Then, $\mathcal{S}$ deletes $eid(f_j)$ from the second-level ADS of all given keywords (in the respective WS_{w_i}s), removes FS_{f_j}, and deletes the cell indexed by $eid(f_j)$ from FS. Finally, she outputs the delete token as $T_d = (eid(f_j), FS[eid(f_j)], id(f_j))$.*

Simulating the modify token *The simulator checks if a hash function key is already assigned to $eid(f_j)$. If not, she first generates a new k-bit random key k'_j and sets $FS[eid(f_j)]=k'_j$. Then, she sends the $FS_{f_j}[t][1]$ values corresponding to the keywords being deleted to the server, and receives the corresponding randomnesses. The leakage $\mathcal{L}_{Mod}(\boldsymbol{f}, f_j) = (eid(f_j), |f_j^{new}|, \{eid(w_{i_t})\}_{t=1}^{t_1}, \{eid(w_{i'_t}), r_{i'_t}\}_{t=1}^{t_2})$ gives information about the keywords being added or deleted. She updates the file accordingly and treats the newly-added keywords as in add token generation. For the keywords to be deleted, she programs the corresponding parts of the WO and FO accordingly, and sends $FO[FS[eid(w_{i'_t})]][r_{i'_t}]$ for all these keywords in the token. These steps bring the FS, WS, FO, WO, FS_{f_j}, and WS_{w_i}s to a consistent and up-to-date state. Finally, she outputs $T_m = (eid(f_j), \{FS_{f_j}[l_t][1], FS_{f_j}[l_t][2]||FS_{f_j}[l_t][3], eid(w_{i_t}), WS_{w_{i_t}}[l'_t][1], WS_{w_{i_t}}[l'_t][2]|| WS_{w_{i_t}}[l'_t][3]\}_{t=1}^{t_1}, \{FS_{f_j}[i'_t][1], FO[FS[eid(w_{i'_t})]][r_{i'_t}]\}_{t=1}^{t_2}$, where l_t, l'_t, and i'_t are indices of the file identifier and its keywords to be added or deleted.*

Answering random oracle queries *In our simulation, add and modify operations always program the random oracle matrices and make them ready for use. Therefore, search and delete operations always find the required random oracle values inside matrices. H^1 random oracle queries with key K and randomness r is always responded with $WO[K][r]$, and that of H^2 is responded with $FO[K][r]$.*

All operations of the simulator are polynomial, making the total running time of the simulator polynomial (since there will be at most polynomially-many adversary queries). Moreover, based on our assumptions, all operations are performed by the simulator in a way that the adversary cannot distinguish them from what the real client outputs.

B Security Proof in the Standard Model

The verifiability is exactly the same. Below, we present the simulation for our construction being private.

Proof 3 ***Overview:*** *The idea and indistinguishability assumptions behind the proof is similar to the random oracle model version. Now, we construct such a PPT simulator $\mathcal{S}$ who adaptively simulates the encrypted files (ciphertexts), the indices, and tokens.*

Initialization *$\mathcal{S}$ uses the keys generated via* `KeyGen` *for building some special internal data structures required during simulation. For each $eid(w_i)$ given by $\mathcal{L}_{Init}$, $\mathcal{S}$ sets $cnt_{w_i} = |\boldsymbol{f}_{w_i}|$, i.e., the number of files containing w_i. For each file identifier $eid(f_j)$ given by $\mathcal{L}_{Init}$, $\mathcal{S}$ sets $cnt_{f_j} = |\boldsymbol{w}_{f_j}|$, i.e., the number of keywords f_j contains. Then, for each w_i, she generates cnt_{w_i}-many l-bit random val_{f_j} values, keeps them in a vector WS_{w_i} of size cnt_{w_i}, and uses them together with their l'-bit location numbers to build a second-level ADS. The roots of these ADSs are used as values together with the respective keys $eid(w_i)$ to build the first-level ADS of the forward index. Moreover, for each f_j, she generates cnt_{f_j}-many $(2l+l')$-bit random val_{w_i} values, stores them in a vector FS_{f_j} of size cnt_{f_j}, and uses them together with their l'-bit location numbers to build a second-level ADS. She uses the roots of these ADSs as values together with the corresponding keys $eid(f_j)$ to build the first-level ADS of the inverted index. She also puts each encrypted file in a DPDP structure and uses their roots as values together with $eid(f_j)$ as their keys to build the first-level ADS of the file index.*

Simulating a single-keyword search token *Using the information in the leakage function $\mathcal{L}_{Srch}(\boldsymbol{f}, w_i) = (eid(w_i), \{eid(f_j)\}_{w_i \in f_j})$, the simulator creates another vector WA_{w_i} of size cnt_{w_i} and stores $\boldsymbol{f}_{w_i}$ in WA_{w_i} in the order specified by $\mathcal{L}_{Srch}$. Then, she outputs the token as $T_s = (eid(w_i), \{WS_{w_i}[t] \oplus WA_{w_i}[t]\}_{t=1}^{cnt_{w_i}})$.*

Simulating a Boolean-search token *The simulator takes similar steps for each keyword in $\mathcal{L}_{BlSrch}$.*

Simulating the add token *Using $\mathcal{L}_{AddDel}(\boldsymbol{f}, f_j) = (eid(f_j), |f_j|, \{eid(w_i)\}_{w_i \in f_j})$, the simulator first updates her local data structures. She sets $cnt_{f_j} = |\boldsymbol{w}_{f_j}|$, stores $\boldsymbol{w}_{f_j}$ in FA_{f_j} and generates cnt_{f_j}-many $(2l + l')$-bit random values, and puts them in FS_{f_j}. For each $eid(w_i) \in \boldsymbol{w}_{f_j}$, she increments the related cnt_{w_i} by one, appends $eid(f_j)$ to the related WA_{w_i}, and generates an l-bit random value $val^i_{f_j}$ to be appended into WS_{w_i}. Finally, she outputs $T_a = (eid(f_j),\ \{t, FS_{f_j}[t], eid(w_{i_t}), cnt_{w_{i_t}}, WS_{w_{i_t}}[cnt_{w_{i_t}}]\}_{t=1}^{cnt_{f_j}})$, where w_{i_t} is the t^{th} keyword in $\boldsymbol{w}_{f_j}$.*

Simulating the delete token *Using $\mathcal{L}_{AddDel}(\boldsymbol{f}, f_j) = (eid(f_j), |f_j|, \{eid(w_i)\}_{w_i \in f_j})$, $\mathcal{S}$ first stores $\boldsymbol{w}_{f_j}$ in FA_{f_j} and sets $cnt_{f_j} = |\boldsymbol{w}_{f_j}|$. Then, for each $eid(w_i) \in \boldsymbol{w}_{f_j}$, she puts NULL in the cells corresponding to $eid(f_j)$ in the related WA_{w_i}. She finally outputs T_d=$(eid(f_j), \{FS_{f_j}[t] \oplus \langle FA_{f_j}[t]||l_t\rangle\}_{t=1}^{cnt_{f_j}})$, where l_t is the location of $FA_{f_j}[t]$ in the corresponding second-level ADS of the forward index, deletes FS_{f_j} and FA_{f_j}, and sets cnt_{f_j}=0. Note that the XOR is performed on the last $(l + l')$ bits of $FS_{f_j}[t]$.*

Simulating the modify token *$\mathcal{S}$ sends the first l-bits of the $FS_{f_j}[t]$ values corresponding to the keywords being deleted to the server, who responds with their location in the related second-level ADS. $\mathcal{L}_{Mod}(\boldsymbol{f}, f_j)$=$(eid(f_j), |f_j^{new}|, addSet$=$\{l_t, eid(w_{i_t})\}_{t=1}^{t_1}, delSet = \{l'_t, eid(w_{i'_t})\}_{t=1}^{t_2})$ gives information about the list and location of keywords being added or removed. $\mathcal{S}$ first updates the file and generates the DPDP update information. Then, if the respective FA_{f_j} does not exist, she builds an empty FA_{f_j} of size cnt_{f_j}, and stores inside it the added and deleted keyword's identifiers. She outputs $T_m = (eid(f_j),\ \{l_t, FS_{f_j}[l_t], eid(w_{i_t}), cnt_{w_{i_t}}, FS_{w_{i_t}}[cnt_{w_{i_t}}]\}_{l_t \in addSet}), \{l'_t, FS_{f_j}[l'_t] \oplus \langle FA_{f_j}[l'_t]||i_t\rangle\}_{l_t \in delSet})$, where i_ts point to the locations of deleted keywords in the related WS_{w_i}. Finally, she updates the corresponding cnt_{w_i}s, cnt_{f_j}s, FS_{f_j}s, WSw_is, FS, and WS accordingly.*

All operations of the simulator are polynomial, making the total running time of the simulator polynomial (since there will be at most polynomially-many adversary queries). Moreover, based on our assumptions, all operations are performed by the simulator in a way that the adversary cannot distinguish them from what the real client outputs.

C HADS Scheme Definition

The (H)ADS scheme is defined as the following polynomial-time algorithms [21]:

$(sk, pk) \leftarrow$ `KeyGen`(1^k) is a probabilistic algorithm run by the client to generate a pair of private and public keys (sk, pk) given as input the security parameter k. The client stores them locally and shares the public key pk with the server.

$(ans, \pi) \leftarrow$ `Certify`(pk, cmd) run by the server to respond to commands coming from the client. It takes the public key pk and command cmd as input. For query commands, it outputs a *verification proof* π enabling the client to verify the validity of the answer ans. For update commands, the ans is null and π is a *consistency* proof enabling the client to update her local metadata.

$\{$`accept`$,$ `reject`$, M'\} \leftarrow$ `Verify`(sk, pk, ans, π, M) run by the client to verify the coming responses. It takes as input the public and private keys (pk, sk), the answer ans, the proof π, and the client's current metadata M, and outputs an acceptance or a rejection signal based on the result of the verification. If the command was update, and the proof is accepted, then the client updates her metadata (to M').

D DPDP Scheme Definition

The DPDP scheme is defined as the following efficient algorithms [18]:

$\{sk, pk\} \leftarrow$ `KeyGen`(1^k) is a probabilistic algorithm run by the client to generate the secret and public key pair (sk, pk), which takes the security parameter as input. The client keeps the secret and public keys, and sends the public key to the server.

$\{e(F), e(info), e(M)\} \leftarrow$ `PrepareUpdate`$(sk, pk, F, info, M_c)$ is an algorithm executed by the client to prepare the file to be stored on the server. It takes as input the secret and public keys, the file F, the definition $info$ of the update to be performed, and the previous metadata M_c, and generates an encoded version of $e(F)$, the information $e(info)$ about the update, and the new metadata $e(M)$. The outputs are sent to the server.

$\{F_i, M_i, M'_c, P_{M'_c}\} \leftarrow$ `PerformUpdate`$(pk, F_{i-1}, M_{i-1}, e(F), e(info), e(M))$ is an algorithm executed by the server upon receipt of an update request. The public key pk, the previous version of the file F_{i-1}, the metadata M_{i-1}, and the outputs of the `PrepareUpdate` function are given as input. It generates the new version of the file F_i, the metadata M_i, along with the metadata and its proof to be sent to the client (M'_c and P'_{M_c}).

$\{$`accept`$,$ `reject`$\} \leftarrow$ `VerifyUpdate`$(sk, pk, F, info, M_c, M'_c, P_{M'_c})$ run by the client to verify the server's response with inputs of `PrepareUpdate` algorithm, M'_c and P'_{M_c}. It outputs an acceptance or a rejection signal.

$\{c\} \leftarrow$ `Challenge`(sk, pk, M_c) is a probabilistic algorithm run by the client to create a challenge to be sent to the server. Given the secret and public keys, and the latest client metadata M_c as input, it generates a challenge c.

$\{P\} \leftarrow$ `Prove`(pk, F_i, M_i, c) run by the server upon receipt of a challenge, given the public key, the latest version of the file and metadata, and the challenge. It generates a proof P to be sent to the client.

$\{$`accept`$,$ `reject`$\} \leftarrow$ `Verify`(sk, pk, M_c, c, P) is run by the client to verify the proof P from the server, given the secret and public keys, the client metadata M_c, the challenge, and the proof P as input. The output is an '`accept`' that ideally means the server keeps storing the file intact, or a '`reject`', otherwise.

E Detailed Construction of Our VDSSE

Listing 1. Our construction

```
Let SKE=(KeyGen,Enc,Dec) be a private-key encryption scheme, FI, II, and FX be HADS schemes with algorithms (KeyGen,Certify,Verify), $F : \{0,1\}^k \times \{0,1\}^{l'} \to \{0,1\}^l$ be a PRF family, and $H^1 : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^l$ and $H^2 : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^{2l}$ be hash function families modeled as random oracles, where $l$ is the length of identifiers and keywords, and $k$ is the security parameter.
KeyGen($1^k$) Generate three random keys as $K_i \leftarrow$ GenPRF($1^k$) for $1 \le i \le 3$, $K_{SKE} \leftarrow$ SKE.KeyGen($1^k$), $(pk_{FI}, sk_{FI}) \leftarrow$
  FI.KeyGen($1^k$), $(pk_{II}, sk_{II}) \leftarrow$ II.KeyGen($1^k$), and $(pk_{FX}, sk_{FX}) \leftarrow$ FX.KeyGen($1^k$). Output $pk = (pk_{FI}, pk_{II}, pk_{FX})$, $sk =$
  $(K_1, K_2, K_3, K_{SKE}, sk_{FI}, sk_{II}, sk_{FX})$, and send $pk$ to the server.
BuildIndex($sk, \mathbf{f}$) // w is the dictionary.
  Building the forward index:
  • For each $w_i \in \mathbf{w}$ :
    – Set $K_{w_i} = F_{K_3}(id(w_i))$ and $K'_{w_i} = F_{K_2}(id(w_i))$.
    – Find the set of file identifiers $\mathbf{f}_{w_i} \subseteq \mathbf{f}$ containing $w_i$.
    – For each $id(f_j) \in \mathbf{f}_{w_i}$:
        ∗ $key_{f_j} = F_{K'_{w_i}}(id(f_j))$.
        ∗ $val_{f_j} = ((id(f_j) \oplus H^1_{K_{w_i}}(r_j)), r_j)$.
    – Build an authenticated skip list, $\mathrm{FI}_{w_i}$, with $(key_{f_j}, val_{f_j})$ as (key, value) and find its root $R_{w_i}$.
  • Construct the first-level authenticated skip list, FI, over $|\mathbf{w}|$-many leaves, using $F_{K_1}(id(w_i))$ as key and $R_{w_i}$ as value at
    each leaf. Call its root $R_{FI}$.
  Building the inverted index and the file index:
  • For each $f_j \in \mathbf{f}$:
    – $c_j =$ SKE.Enc$_{K_{SKE}}(f_j)$.
    – $K_{f_j} = F_{K_3}(id(f_j))$ and $K'_{f_j} = F_{K_2}(id(f_j))$.
    – Find the set of distinct keyword identifiers $\mathbf{w}_{f_j} \subseteq \mathbf{w}$ that appear in $f_j$.
    – For each $id(w_i) \in \mathbf{w}_{f_j}$:
        ∗ $key_{w_i} = F_{K'_{f_j}}(id(w_i))$.
        ∗ $val_{w_i} = ([F_{K_1}(id(w_i))||key_{f_j}] \oplus H^2_{K_{f_j}}(r_i), r_i)$.
    – Build an authenticated skip list, $\mathrm{II}_{f_j}$, with $(key_{w_i}, val_{w_i})$ as (key, value) and find its root $R_{f_j}$.
    – Build a second-level ADS according to DPDP (or FlexDPDP) for $c_j$ and find its root $RDP_{f_j}$.
  • Construct the first-level authenticated skip list, II, over $|\mathbf{f}|$-many leaves, using $F_{K_1}(id(f_j))$ as key and $R_{f_j}$ as value at each
    leaf. Call its root $R_{II}$.
  • Construct the first-level authenticated skip list, FX, over $|\mathbf{f}|$-many leaves, using $F_{K_1}(id(f_j))$ as key and $RDP_{f_j}$ as value at
    each leaf. Call its root $R_{FX}$.
  • Set $\mathbf{c} = (c_1, c_2, ..., c_n)$.
  • Set $\mathcal{I} = (FI, \{FI_{w_i}\}_{w_i \in \mathbf{w}}, II, \{II_{f_j}\}_{f_j \in \mathbf{f}}, FX, \{RDP_{f_j}\}_{f_j \in \mathbf{f}})$.
  • Set $M = (R_{FI}, R_{II}, R_{FX})$. // The local metadata
  • Output $(\mathcal{I}, \mathbf{c}, M)$. // $\mathcal{I}$ and c are sent to the server who stores them.
Dec($sk, c_j$) Output $f_j =$ SKE.Dec$_{K_{SKE}}(c_j)$.
SearchToken($sk, M, w_i$) Output $T_s = (F_{K_1}(id(w_i)), K_{w_i})$.
Search($\mathcal{I}, \mathbf{c}, T_s$) .
  • Interpret $T_s$ as $(K^1, K^2)$.
  • $(Node, \pi_1) =$ FI.Certify($pk_{FI}$, ‘Challenge $K^1$’). // Find a node through the key $K^1$.
  • If $Node = NULL$ then return $(\{\}, \pi_1)$. // The non-membership proof.
  • Find the second-level ADS whose root is $Node.val$.
  • For each leaf of this second-level ADS who stores $(d_j, r_j)$:
    – Decrypt each leaf value as $id(f_j) = d_j \oplus H^1_{K^2}(r_j)$.
    – Put the ciphertext with $id(f_j)$, i.e., $c_j$, in $\mathbf{c}_w$.
  • $\pi_2 =$ FX.Certify($pk_{FX}$, ‘Challenge $\mathbf{c}_w$’). // Integrity proof generation by the file index.
  • Return $(\mathbf{c}_w, \pi_1||\pi_2)$.
BooleanSearchToken($sk, M, \phi(w_1, ..., w_t)$) .
  • Set $T_b = \phi(w_1, ..., w_t)$.
  • Replace each $w_i \in T_b$ with $(F_{K_1}(id(w_i)), K_{w_i})$.
  • Output $T_b$.
BooleanSearch($\mathcal{I}, \mathbf{c}, T_b, ch$) .
  • Compute file identifiers satisfying $T_b$ and corresponding proof, $\pi_1$, as in Section 4.5.
  • Put in $\mathbf{c}_w$ ciphertexts of the file identifiers found.
  • $\pi_2 =$ FX.Certify($pk_{FX}$, ‘Challenge $\mathbf{c}_w$’). // Integrity proof generation by the file index.
  • Output $(\mathbf{c}_w, \pi_1||\pi_2)$.
```

Listing 1. Our construction (Cont'd.)

```
AddToken($sk, M, f_j$) // $f_j$ has unique keywords $(w_1, ..., w_t)$.
  • $c_j = \mathtt{EncSPE}_{K_{SKE}}(f_j)$.
  • $T_a = (F_{K_1}(id(f_j)), \{key_{w_i}, val_{w_i}, F_{K_1}(id(w_i)), key^i_{f_j}, val^i_{f_j}\}^t_{i=1})$.
  • Prepare the corresponding file update information, $infoAdd_{c_j}$, according to the HADS.
  • Output $(T_a, infoAdd_{c_j}, c_j)$.
Add($\mathcal{I}, \mathbf{c}, T_a, infoAdd_{c_j}, c_j$) .
  • $\mathbf{c}' = \mathbf{c} \wedge c_j$.
  • Interpret $T_a$ as $(K^I_{f_j}, \{K^I_{w_i}, V^I_{w_i}, K^F_{w_i}, Ki^F_{f_j}, Vi^F_{f_j}\}^t_{i=1})$.
  • FX.Certify($pk_{FX}, infoAdd_{c_j}$). // Put $c_j$ in a second-level ADS whose root will be $RDP_{f_j}$.
  • $\pi_1 =$ FX.Certify($pk_{FX}$, 'Add $(K^I_{f_j}, RDP_{f_j})$'). // Tie it to $f_j$ in the first level of FX.
  • II.Certify($pk_{II}$, 'Build $\{(K^I_{w_i}, V^I_{w_i})\}$'). //Create an ADS with all keywords whose root is $R_{f_j}$.
  • $\pi_2 =$ II.Certify($pk_{II}$, 'Add $(K^I_{f_j}, R_{f_j})$'). // Tie it to $f_j$ in the first level of II.
  • For each tuple in $\{K^F_{w_i}, Ki^F_{f_j}, Vi^F_{f_j}\}^t_{i=1}$:
    – $Node =$ FI.Certify($pk_{FI}$, 'Challenge $K^F_{w_i}$'). // Locate $w_i$ in the first level of the FI.
    – If $Node = NULL$ // The keyword does not exist in the forward index.
        ∗ FI.Certify($pk_{FI}$, 'Build $(Ki^F_{f_j}, Vi^F_{f_j})$'). // Create an ADS whose root is $R_{w_i}$.
        ∗ $\pi_3 = \pi_3||$FI.Certify($pk_{FI}$, 'Add $(K^F_{w_i}, R_{w_i})$'). // Tie it to $w_i$ in the first level of FI.
    – Else
        ∗ FI.Certify($pk_{FI}$, 'Add $(Ki^F_{f_j}, Vi^F_{f_j})$').// Add the file id to the second-level ADS found.
        ∗ $\pi_3 = \pi_3||$FI.Certify($pk_{FI}$, 'Modify $(K^F_{w_i}, R'_{w_i})$'). // Apply the update in the first level.
  • Output (the modified index $\mathcal{I}', \mathbf{c}', \pi_1||\pi_2||\pi_3$).
DeleteToken($sk, M, id(f_j)$) :
  • Output $T_d = (F_{K_1}(id(f)), K_{f_j}, id(f_j))$.
Delete($\mathcal{I}, \mathbf{c}, T_d$) :
  • Interpret $T_d$ as $(K^1, K^2, id(f_j))$.
  • $\mathbf{c}' = \mathbf{c} \setminus c_j$.
  • $(Node, \pi_1) =$ II.Certify($pk_{II}$, 'Challenge $(K^1)$'). // Find the leaf node with key $K^1$ on II.
  • If $Node = NULL$ then return $\pi_1$. // The non-membership proof.
  • Find the second-level ADS whose root is $Node.val$.
  • For each leaf of this second-level ADS who stores $(d_j, r_j)$.
    – Decrypt each leaf value as $(id(w_i), K_{w_i}) = d_j \oplus H^2_{K^2}(r_j)$.
    – FI.Certify($pk_{FI}$, 'Delete $(id(w_i), K_{w_i})$'). // Delete a second-level node with key $K_{w_i}$ on FI.
    – $\pi_2 = \pi_2||$FI.Certify($pk_{FI}$, 'Modify $(id(w_i), R'_{w_i})$'). // Deletion affects the first level of FI.
  • $\pi_3 =$ II.Certify($pk_{II}$, 'Delete $(K^1)$'). // Delete $K^1$ (with the second-level ADS) from II.
  • $\pi_4 =$ FX.Certify($pk_{FX}$, 'Delete $(K^1)$'). // Delete $K^1$ (with the second-level ADS) from FX.
  • Output (the modified index $\mathcal{I}', \mathbf{c}', \pi_2||\pi_3||\pi_4$).
ModifyToken($sk, M, id(f_j), m$) :
  • $T_m = (F_{K_1}(id(f_j)), \{key_{w_i}, val_{w_i}, F_{K_1}(id(w_i)), key^i_{f_j}, val^i_{f_j}\}^{t_1}_{i=1}, \{key_{w_i}, H^2_{K_{f_j}}(r_i)\}^{t_2}_{j=1})$, where $t_1$ and $t_2$ are the number
    of keywords being added and deleted.
  • Prepare the corresponding update information, $infoMod_{f_j}$, according to the HADS.
  • Output $(T_m, infoMod_{f_j})$.
Modify($\mathcal{I}, \mathbf{c}, T_m, infoMod_{f_j}$) :
  • Update $f_j$ and achieve $\mathbf{c}'$.
  • Interpret $T_m$ as $(K^I_f, \{K^I_{w_i}, V^I_{w_i}, Ki^F_w, K'^F_f, Vi^F_f\}^{t1}_{i=1}, \{K'^I_{w_i}, h'^I_{w_i}\}^{t_2}_{j=1})$.
  • FX.Certify($pk_{FX}, infoMod_{c_j}$). // Update the second-level ADS whose root is $RDP_{f_j}$.
  • $\pi_1 =$ FX.Certify($pk_{FX}$, 'Modify $(K^I_f, RDP'_{f_j})$'). // Update the first-level ADS of FX.
  • $\pi_2 =$ perform the add part as in the Add algorithm.
  • $\pi_3 =$ perform the delete part as in the Delete algorithm.
  • Output (the modified index $\mathcal{I}', \mathbf{c}', \pi_1||\pi_2||\pi_3$).
Verify($sk, pk, M, \mathbf{c}_w, \pi$) :
  • Interpret $\pi$ as $\pi_{FI}||\pi_{II}||\pi_{FX}$.
  • $ver_{FI} = FI.$Verify($sk_{FI}, pk_{FI}, \mathbf{c}_w, \pi_{FI}, R_{FI}$).
  • $ver_{II} = II.$Verify($sk_{II}, pk_{II}, \mathbf{c}_w, \pi_{II}, R_{II}$).
  • $ver_{FX} = FX.$Verify($sk_{FX}, pk_{FX}, \mathbf{c}_w, \pi_{FX}, R_{FX}$).
  • If at least one of them shows rejection, output 'reject'.
  • If the operation was Search or BooleanSearch: check the list of received files against the identifiers in the proof, and output
    'reject' for any mismatch.
  • Update local state $M$ accordingly and output 'accept'.
```

F The Boolean Formula Proof Generation Algorithm

Algorithm 1: CompBooleanFID, run by the server.

```
Input: $t$ second-level ADSs: $ADS_1 ... ADS_t$, their keys: $K_1 ... K_t$, and literals: $N_1 ... N_t$.
Output: List of file identifers $FID$, and the proof $\pi$.
  // Assume that ADSs are given in increasing order of the number of leaves.
  $\pi = \{\}$;
  Decrypt all values in leaves of the $ADS_1$.
  if $N_1 \;==\; '\neg'$ then
      FID = $\mathbf{f} - \{f_i | f_i \in ADS_1\}$.
  else
      FID = $\{f_i | f_i \in ADS_1\}$.
  $\pi = \{(f_i, level) | f_i \in ADS_1\}$.
  for each $ADS_i \in \{ADS_2, ..., ADS_t\}$ do
      $\pi_i = FID' = \{\}$;
      // Each $ADS_i$ is encrypted with a different key.  Decrypt them before comparison.
      $FID_i = \{f_j | f_j = Dec(K_i, f'_j) \wedge \; f'_j \in ADS_i\}$.
      for each $f_j \in FID$ do
          if $f_j \in FID_i$ then
              $\pi' = ADS_i.GenMembershipProof(f'_j)$;
              if $N_i \;\neq\; '\neg'$ then
                  $FID' = FID' \cup f_j$;
          else
              $\pi' = ADS_i.GenNonMembershipProof(f'_j)$;
              if $N_i \;==\; '\neg'$ then
                  $FID' = FID' \cup f_j$;
          $\pi_i = \pi_i || \pi'$;
      $\pi = \pi || \pi_i$;
      $FID = FID'$; //Consider only these selected identifiers for the next round, not all in FID.
      Important:  the FID size is decreasing as we progress.
return $(FID, \pi)$;
```