Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions

Javad Ghareh Chamani
HKUST

Dimitrios Papadopoulos
HKUST

Mohammadamin Karbasforushan
UC Santa Cruz

Ioannis Demertzis
UC Santa Cruz

Abstract

We focus on the problem of Dynamic Searchable Encryption (DSE) with efficient (optimal/quasi-optimal) search in the presence of deletions. Towards that end, we first propose OSSE, the first DSE scheme that can achieve asymptotically optimal search time, linear to the result size and independent of any prior deletions, improving the previous state of the art by a multiplicative logarithmic factor. We then propose our second scheme LLSE, that achieves a sublogarithmic search overhead ($\log\log i_w$, where i_w is the number or prior insertions for a keyword) compared to the optimal achieved by OSSE. While this is slightly worse than our first scheme, it still outperforms prior works, while also achieving faster deletions and asymptotically smaller server storage. Both schemes have standard leakage profiles and are forward-and-backward private. Our experimental evaluation is very encouraging as it shows our schemes consistently outperform the prior state-of-the-art DSE by 1.3-6.4$\times$ in search computation time, while also requiring just a single roundtrip to receive the search result. Even compared with prior simpler and very efficient constructions in which all deleted records are returned as part of the result, our OSSE achieves better performance for deletion rates ranging from 45-55%, while the previous state-of-the-art quasi-optimal scheme achieves this for 65-75% deletion rates.

1 Introduction

Searchable encryption (SE) [17, 59] allows a user to upload her dataset to an untrusted server in encrypted form, while maintaining the ability to search over it without revealing raw data to the server. Since its introduction by Song et al. [59], it has become a heavily studied research topic with numerous works that try to improve its security [21, 38, 41, 54], efficiency [6, 20, 24, 25, 51], expressiveness of supported types of queries [11, 12, 15, 22, 23, 28, 37, 49], and support for multiple users [34, 56, 57, 64, 65]. One common denominator in SE research has been the focus on highly practical schemes that can be used in real-world applications and scale to large datasets, at the cost of a clearly defined *leakage of information* about the dataset to the server such as whether queries are repeated, the size of the returned result, and when the same results are returned. Partly due to this emphasis on efficient performance, SE has been used in a variety of settings starting from simple text keyword search [17, 59], and including encrypted relational [12, 39], NoSQL [55], or graph [30, 47, 49] databases. Consequently, SE has been proposed for applications such as annotated image search [3], encrypted email clients [50], and very recently for maintaining a secure gun registry [42].

Our focus in this work is *dynamic searchable encryption (DSE)* [8, 43, 44], that is, schemes that allow the user to efficiently modify the dataset, without having to re-run a costly setup process from scratch. More specifically, we are interested in DSE with efficient search performance in the presence of deletions. Ideally, the search time for a query should be *optimal*, i.e., proportional to the result size itself, *independently of the number of prior deletions that affect this result*. For instance, consider an extremely simple example for a DSE keyword search over a textual dataset, returning which files contain it. If the keyword appears in 100 files but subsequently 90 of them are deleted, the search overhead should only be proportional to its actual returned result (i.e., 10). Given the wide range of potential SE applications, this can make a big difference in the system's performance.

In particular, in a graph database deleting a node immediately triggers deletions of all its adjacent edges. For a relational database (likewise, for streaming systems [4, 35] and pure key-value storage [1]), which use a (LSM-like) key-value store (or an encrypted store in the case of DSE) as a storage layer [29], a single deletion/update of a tuple/value will produce a multiplicative number of deleted/cancellation tuples—at least one for each attribute. These deleted tuples can later interleave with future search queries affecting the search performance. Such issues have also been observed recently by [58]; that work considers settings in which "low-level" deletes are triggered not only by users' direct actions, and studies scenarios of delete-intensive query workloads.

Table 1: Comparison with prior DSE with quasi-optimal search. N is the upper bound on the (w,id) pairs and D total number of deletions, $|W|$ is the number of distinct keywords, and for each keyword, i_w and d_w are the number of insertions and deletions, $a_w = i_w + d_w$ is the total number of updates, and $n_w = i_w - d_w$ is the number of non-deleted (w,id) pairs. RT is the number of roundtrips for completing a search query. We assume oblivious maps are instantiated with [63]. For QOS, we refer to the version from [19] with counters locally stored at the client.

Scheme	Search	Insert	Delete	Search RT	Server Storage	BP
SPS [60]	$O(\min\{a_w + \log N, n_w \log^3 N\})$	$O(\log^2 N)$	$O(\log^2 N)$	1	$O(N)$	✗
Orion [13]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(N)$	I
Horus [13]	$O(n_w \log d_w \log N + \log^2 \lvert W\rvert)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(N)$	III
QOS [19]	$O(n_w \log i_w)$	$O(\log^3 N)$	$O(\log^3 N)$	2	$O(N)$	III
OSSE (Sec. 4)	$O(n_w + \log i_w)$	$O(\log^2 N)$	$O(\log^3 N)$	1	$O(N + D \log N)$	III
LLSE (Sec. 5)	$O((n_w + \log i_w) \cdot \log\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	1	$O(N)$	III

Prior DSE with (quasi-)optimal search. The majority of DSE schemes in the literature (e.g., [9, 13, 19, 27]) adopt a "lazy deletion" policy, treating deletions as actual entries on their own behalf (similar to the LSM-like policy mentioned above). In subsequent searches, the client retrieves all relevant inserted entries and "deletion entries" and filters out the actual result. While this leads to a conceptually simple search process, it can have significant adverse impact on the system's search performance, as motivated in our discussion above. That said, a small number of works focus on the same goal as we do and we provide an overview of their asymptotic performance in Table 1, starting from Stefanov et al. [60] that proposed the first DSE where the search performance is always *quasi-optimal*, i.e., within a poly-logarithmic factor from the result size. Unfortunately, their construction has the drawback that it reveals to the server the id's of records that contained the searched keyword but have since been deleted[1], which may not be acceptable in some applications (e.g., as related to the *right-to-erasure* of EU GDPR article 17).

Formally this property has been defined as *backward privacy* in [9], distinguishing it from *forward privacy* [14, 60] that minimizes information leakage during updates. Focusing on works that achieve both forward and backward privacy, the best such scheme (QOS from [19]) requires $O(n_w \log i_w)$ for a search for keyword w with result size n_w after i_w relevant entry insertions.[2] Previous works (Orion and Horus from [13]) embed the entire dataset inside an oblivious index [18, 31, 61, 63]. If this is instantiated with the state-of-the-art oblivious indexes [5], the result would be asymptotically the same as QOS, i.e., a logarithmic factor slower than the optimal. Interestingly, there exist very strong lower-bound results [31, 48, 53] that show that relying solely on oblivious indexes one *cannot* hope to further improve search performance asymptotically.

Motivated by this, we ask whether it is possible to design a forward-and-backward private DSE with search performance that is asymptotically within a sublogarithmic factor from the optimal $O(n_w)$, or indeed with *optimal* search performance.

This work. We present the first DSE schemes that answer the above question in the affirmative. Our first construction OSSE (Section 4) is the only existing DSE that, for large enough result sizes, requires an optimal number of $O(n_w)$ operations for search queries (and for very small hidden constants). Its main drawback is a somewhat increased storage, as the size of the encrypted storage grows logarithmically with the total number of deletions (that said, in our experimental evaluation we show that this is not a prohibitive factor in practice). Our second scheme LLSE requires $O(n_w \log\log i_w)$ operations for searches with large enough results. While this is weaker than our first result, LLSE still outperforms the prior state of the art QOS by a factor or $O(\log i_w / \log\log i_w)$, while also achieving better deletion performance than OSSE and QOS, and asymptotically optimal storage (see Table 1). Moreover, both our schemes retrieve the search result with a single roundtrip.

At the core of our results lies a carefully chosen arrangement of entries in the encrypted dataset as conceptual keyword binary trees (see Figure 3 and discussion in Section 4.1). With insertions lying at the leaf level, subsequent deletions "prune" this tree to minimize the number of traversed such nodes during a search. The key challenge is how to manipulate these trees without revealing significant information to the server. Interestingly, our schemes use an oblivious map to store tree data, however, they only access it during updates (using retrieved information as "guidelines" to prepare future searches) thus making searches extremely fast and non-interactive.

Our contributions can be summarized as follows:

1. We propose OSSE, the first DSE that can achieve asymptotically optimal search performance (for $n_w \geq \log i_w$), independently of prior deletions, and LLSE, a DSE with search performance within an asymptotic loglog factor from the optimal, that also achieves faster deletions and linear storage. For both schemes, we prove their security for standard leakages and show they are forward-and-backward private (BP-III).

2. We propose a series of performance optimizations (Section 6) and report on the efficiency of prototype imple-

[1] Having been inserted after the last search for this keyword; otherwise, the server already knows these id's as information leakage during that search.

[2] As in all prior works referenced here, when referring to asymptotics we implicitly assume map lookups are constant-time operations, e.g., via dynamic perfect hashing [26].

mentations of our schemes (Section 7.1) with encouraging results (e.g., optimized OSSE takes $< \mathbf{1ms}$ of computation time to retrieve a result of 100 id's and $< \mathbf{10ms}$ for a result of 20K id's, from a database of 1M entries).

3. We compare the performance of our schemes vs. existing quasi-optimal ones (Section 7.2) for different settings (variable deletion rates, for both random and "structured" deletions). Both our constructions consistently outperform QOS by roughly **1.3-4**× but for large result sizes and large deletion rates the improvement is up to **6.4**×.

4. We also compare our schemes with conceptually simpler DSE with "lazy" deletion (Section 7.2). As expected, as deletion rates increase our schemes outperform SD_d from [19]. Concretely, for OSSE this happens for rates **45-55**%, and our fully optimized OSSE with result caching, in the best case, can outperform SD_d even after a single deletion!. Surprisingly, for large enough sizes this carries over even for a "succinct-client-storage" version of our schemes albeit for larger deletion rates.

Related Work. In our security formulation, we use the standard SE definition of [17] extended to the dynamic setting in [60]. The QOS scheme from Demertzis et al. [19] is currently the best existing quasi-optimal DSE and we benchmark our constructions' performance with it. In [19], the authors propose two versions of QOS with small $O(1)$ and large $O(|W|)$ client storage. We mainly compare our schemes with the second one, but in Section 7.2 we measure performance in both settings. From a technical viewpoint, QOS also encodes entries in keyword trees, however, the way they are maintained cannot achieve our target of sublogarithmic overhead. Moreover, the design of OSSE, allows us to benefit from "early" stop while parsing the tree if empty nodes are found, whereas QOS needs to keep traversing the layers.

Regarding "lazy" deletion schemes, there exist several candidates for performance comparison, e.g., from [9, 13, 19]. Among them, Mitra (with the recently proposed optimizations of [16]) and SD_d stand out. Although they have very similar performance, we chose the latter since, as shown in [19], it can give slightly faster searches, and it is the only one with succinct client storage without performing oblivious map accesses during searches. One point to note is that the way all existing DSE schemes implement "lazy" deletion is by storing a separate record for each deletion. This is unlike "lazy" deletion in plaintext databases where the deletion is usually marked in-place (e.g., using a special flag). Storing this entry separately is necessary; touching the same location in the encrypted index during an update as a previously accessed one would violate forward privacy (see Section 2). Consequently, when using such a scheme the server observes the result size "growing" after deletions, whereas in our schemes it "shrinks." In both cases, this is already captured by the standard leakage profile used in the literature (see Section 2 and Appendix B.1). We also note many of these DSE schemes use similar building blocks as ours (encrypted maps and oblivious indexes). However, the way we use them to embed our "shrinking" keyword trees (Section 4.1) is what allows us to match practical performance with theoretical improvements, achieving the first scheme with optimal search.

Another recent scheme that falls under this category is Aura [62] that also achieves one-round interaction for search, same as our schemes. However, close inspection of the scheme's search process [62, Alg. 1] shows that the server extracts all insertion *and deletion* entries for keyword w before "filtering out" the latter so its search is at least linear to a_w. Finally, recent work by Patel et al. [53] shows it is fundamentally impossible (in the leakage cell probe model) to achieve sublogarithmic search overhead if one requires searches and updates to be independently simulatable without state "carrying" across them during simulation. In relation to this, our schemes are positive evidence that it is possible to overcome this lower bound (even in the more stringent setting with deletions) if one can settle for independent update simulation (guaranteed by our forward privacy) whose state is then available for subsequent searches.

2 Preliminaries

We denote by λ a security parameter and by $\nu(\lambda)$ a negligible function in λ. PPT stands for probabilistic polynomial-time. Our protocols are executed between two parties, a client and a server. The notion $P(x;y)$ denotes a (possibly multi-round) protocol execution where x and y are the client's input and the server's input, respectively.

We consider a collection of F files/documents (this is an abstraction that can possibly capture other data types, e.g., semi-structured data, database records, etc.) with identifiers $id_1, \ldots, id_F$, each of which contains textual keywords from a given alphabet Λ. The database DB consists of keywords and file identifiers such that $(w, id) \in DB$ if and only if the file id contains keyword w. Let W denote the set of all keywords in DB, $|W|$ its cardinality, and N an upper bound on the total number of keyword/document pairs. Then, $DB(w)$ corresponds to the set of documents containing keyword w.

We rely on classic cryptographic tools, such as pseudorandom functions and standard symmetric-key encryption. For completeness, we provide their descriptions in Appendix A.

Oblivious Maps. A map (dictionary) is a data structure that maps addresses (keys) to values and provides read (get) and write (put) access methods. An *oblivious map (OMAP)* is a privacy-preserving version of a regular map that hides the type and content of operations. An OMAP consists of three procedures: (i) *OMAP*.Setup for initializing the data structure, (ii) *OMAP*.put to add/overwrite a key/value pair, and (iii) *OMAP*.get to retrieve the value for a given key. Intuitively, all equal-length sequences of data accesses (get/put) are indistinguishable from each other, even for an adversary that stores the data structure itself (without knowing the secret key). More concretely, this interaction can be simulated given only the number of operations (see [63] for the detailed security formulation). In our schemes, we use the popular OMAP of Wang et al. [63] which stores an AVL-tree inside a PathORAM [61] oblivious array. For a map with capacity N, each access takes $O(\log^2 N)$ operations and $O(\log N)$ roundtrips. We refer interested readers to [63] for additional details.

3 Dynamic Searchable Encryption

A *dynamic symmetric searchable encryption scheme (DSE)* Σ consists of algorithm Setup, and protocols Search and Update that are executed between a client and a server.

- $\mathrm{Setup}(1^{\lambda}, N)$ returns (K, σ, EDB) where K is the client's secret key, σ its local state, and EDB is an (empty) encrypted database, initialized for capacity N, sent to the server.
- $\mathrm{Search}(K, w, \sigma; EDB)$ is a (possibly interactive) protocol for retrieving $DB(w)$, and may also modify K, σ, EDB.
- $\mathrm{Update}(K, op, id, w, \sigma; EDB)$ is a (possibly interactive) protocol for inserting/removing a document-keyword entry (id, w) to/from the database. Operation op is either add or del and the protocol may modify K, σ and EDB.

Here, we mostly follow the API description of [8,9,13]. Using these procedures, the client runs Setup for an empty database, followed by up to N executions of insertions to "populate" EDB. Other works [27,45] propose different but functionally equivalent definitions for Update (e.g., inserting or deleting an entire document, which can be decomposed to multiple calls of Update). Finally, the above definition isolates the goal of retrieving only $DB(w)$ during Search, i.e., the id's of all files/documents containing w. In some applications, the client would like to retrieve the actual documents; this is omitted from our model as it can always be done with an extra round of interaction after the completion of Search.

At a high level, a DSE is correct if the returned $DB(w)$ result is correct for every query and the given sequence of prior updates. The security of a DSE scheme is parametrized by a leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ that captures the information that is revealed to server during the execution of the different processes. $\mathcal{L}^{Stp}$ corresponds to leakage during setup, $\mathcal{L}^{Srch}$ during a search operation, and $\mathcal{L}^{Updt}$ during updates. Commonly encountered types of leakage in the relevant literature, that also occur with our schemes, consist of: (i) *search pattern* that reveals which searches are related to the same w, (ii) *volume pattern*, i.e., $|DB(w))|$ during a search for w, and (iii) *database size* leakage during setup (in our case, the upper bound N). Another very important type of leakage is *access pattern* that reveals the actual contents of $DB(w)$ when w is searched for. This is inherently unavoidable when the actual documents need to be retrieved as explained above (unless stored with "costly" oblivious storage). Here, we focus on retrieving $DB(w)$. Thus our schemes do not directly leak this but, when used in an application that retrieves the actual documents, this has to be taken into account.

Informally, a secure DSE scheme with leakage $\mathcal{L}$ should reveal nothing about the database DB other than this leakage. This is formally captured by a standard real/ideal experiment with two games $\mathsf{Real}^{\mathsf{DSE}}$, $\mathsf{Ideal}^{\mathsf{DSE}}$ presented in Figure 8 in Appendix B, using the following formulation from [60].

Definition 1. *A DSE scheme Σ is adaptively-secure with leakage function $\mathcal{L}$, iff for any PPT adversary Adv issuing polynomially many queries q, there exists a stateful PPT simulator $\mathsf{Sim} = (SimInit, SimSearch, SimUpdate)$ such that $|\Pr[\mathsf{Real}^{\mathsf{DSE}}_{\mathsf{Adv}}(\lambda, q) = 1] - \Pr[\mathsf{Ideal}^{\mathsf{DSE}}_{\mathsf{Adv},\mathsf{Sim},\mathcal{L}}(\lambda, q) = 1]| < \nu(\lambda)$.*

As explained above, many DSE schemes store deletions as actual entries in EDB and filter them to retrieve the actual result during searches. Ghareh Chamani et al. [13], proposed the following definition for schemes that avoid this and achieve search time close to the optimal $O(n_w)$.

Definition 2. *A DSE scheme Σ has* optimal *(resp.* quasi-optimal*) search, if* Search *takes $O(|DB(w)|)$ (resp. $O(|DB(w)| \cdot \mathrm{polylog}(N))$) operations.*

Forward & Backward Privacy. Forward privacy [14, 60] limits the information revealed due to updates in EDB. Informally, a scheme is *forward private* if it is not possible to relate an update to previous operations, *when it takes place*. E.g., it should be impossible to tell whether an insertion is for a new keyword or a previously existing/searched one. Previous works (e.g., [66]) have shown the potential of forward privacy to thwart certain types of leakage-abuse attacks.

Definition 3. *An $\mathcal{L}$-adaptively-secure DSE scheme that supports addition/deletion of a single keyword is forward private iff the update leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'^{Updt}(op, id)$ where $\mathcal{L}'$ is a stateless function, op is insertion or deletion, and id is a file identifier.*

Backward privacy [9] specifies the server should not be able to learn the id's of documents that contained w, if they have since been deleted. Clearly, this is only meaningful if a search for w did not take place prior to this deletion. Bost et al. [9] proposed three different "flavors" of backward privacy with varying leakage patterns, all of which achieve the minimum requirement of hiding id's of such deleted entries. Here, we only aim for BP-III, according to the following definition from [9] (the involved leakage functions are defined in Appendix B.1).

Definition 4. *An $\mathcal{L}$-adaptively-secure DSE scheme has backward privacy (BP-III), iff $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}^{'}(op, w)$ and $\mathcal{L}^{Srch}(w) = \mathcal{L}^{''}(TimeDB(w), DelHist(w))$, where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless functions.*

4 Optimal-Search DSE (OSSE)

We are now ready to present our first construction OSSE. We first describe the main idea of maintaining a conceptual binary tree to store the entries for each keyword w and then we provide details about the algorithms of the scheme.

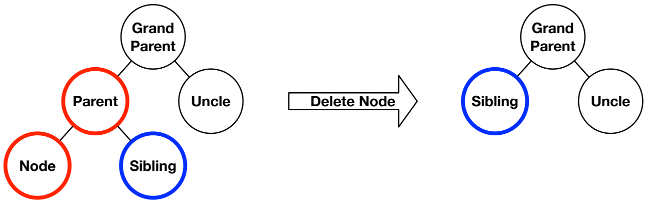


Figure 1: OSSE: Deletion pruning

4.1 Labeled Binary Trees for Stored Entries

In our schemes, we store entries of the form (w, id) such that keyword w appears in document (more generally, record) id. Conceptually, each such entry is mapped to a leaf of a complete binary tree T_w with N leafs (where N is an upper bound for the number of total entries in the dataset), in *chronological order* from left to right. The first entry for w is stored at the leftmost leaf, etc. We consider a function $\mathsf{lab}(v)$ that maps each tree node v to a unique numerical label, as follows. First, leafs are ordered $1, ..., N$ left-to-right, then the remaining nodes are ordered bottom-to-top and left-to-right. E.g., nodes directly above the leafs are labelled starting from the left as $N+1, \dots, 3N/2$, as shown in Figure 3. In the following, we often refer to tree nodes directly by their labels. Note that we use one tree for each keyword but the schemes do not require storing and maintaining "complete" trees (that would make the storage $N \cdot |W|$)—these are just "conceptual" encodings of the N actual (w, id) entries spread among keywords.

Having explained T_w, extracting the result $DB(w)$ after i_w insertions can be achieved trivially by starting from the Minimum Covering Subset (MCS) of nodes for the leafs in $[1, i_w]$. The MCS consists of the smallest set of nodes whose leaf descendants are exactly the leafs labeled $1, \dots, i_w$. E.g., in Figure 3(top), the MCS of $[1, 15]$ is $\{29, 27, 23, 15\}$. Following the edges downwards from the MCS, this process will parse in total $< 2i_w$ nodes. However, recall that our goal is to do better than this, in the presence of deletions. When a deletion takes place, we "remove" from T_w the corresponding leaf node v *and its parent pp*, effectively "pruning" the tree. In practice, we do this by setting the leaf's sibling *sib* as the direct child (left or right, depending on the tree topology) of the grandparent *gp* of v, skipping its parent *pp*. We illustrate this in Figure 1. Handling deletions in this way, leads to the following result which we prove as part of our analysis: Given a sequence of i_w insertions and $d_w \leq i_w$ deletions, the number of nodes of T_w below the MCS that need to be accessed to extract $DB(w)$ is $< 2 \cdot |DB(w)|\ (= n_w)$.

Intuition: From trees to a secure index. Next, we will use these keyword trees to build the secure index for our DSE. The "classic" approach followed by many prior works [8, 9, 11, 11, 13, 19, 27] is to take each entry (in our case, its corresponding tree node) and store it in an *encrypted map*. This is a standard key-value hashmap but: (i) keys (positions) are computed with a pseudorandom hash function, and values are encrypted in a way that only reveals them to the server during searches (and for leaf nodes, not even then). Although this approach gives us the desired efficiency benefit, in our case, it introduces a subtle security issue. The same node may be accessed repeatedly during deletions. E.g., in Figure 3(bottom), deleting "cousin" leafs 10 and 11 requires "touching" their grandparent 27 twice. Even though the value in the hashmap is encrypted, the server can clearly see the same *position* is accessed both times, which unfortunately *violates forward privacy*.

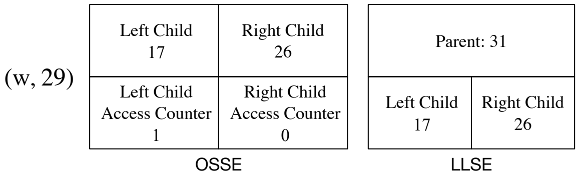


Figure 2: OM_{Tree} data structure of OSSE and LLSE, with node 29 in Figure 3(middle) as an example.

To avoid this issue, in our schemes we use the following trick. First, we give each T_w node its own increasing *access counter*, initially set to 0 and incremented by 1 each time the node is modified. Next, when pseudorandomly computing the new encrypted map position for this node after it is accessed during an update, we also include the incremented access counter (and a freshly encrypted value). Thus, a new write to update the information of an existing tree node is indistinguishable from random and cannot be linked to any previous operation. The remaining challenge is how can the user know these access counters, which is necessary both for searches and for updates. We solve this by storing node information also in an oblivious index (see Figure 2) that, crucially, is accessed *only* during updates. Then during searches, our two schemes follow a different strategy: the first stores node access counters in their parent nodes, whereas in the second the server "finds" the correct access counter by performing a binary search. In the next parts, we present in detail all the data structures used in our DSE and the details of our algorithms.

4.2 OSSE Data Structures & Algorithms

We are now ready to describe OSSE in detail. First, we present the different data structures that we use to maintain and manipulate the trees T_w for the different keywords.

- $\underline{OM_{Tree}}$ is an oblivious map that maps pairs of keywords and node labels $(w, \mathsf{lab}(v))$ to that node data structure v containing the following: $(v.left$, $v.leftAcc$, $v.right$, $v.rightAcc)$. These are the label of the left child of v with its access counter, and the label of the right child of v with its access counter (see Figure 2). This oblivious map is accessed during updates (but not searches) to change T_w.
- $\underline{OM_{Del}}$ is an oblivious map that maps pairs of keywords and document identifiers (w, id) to $\mathsf{lab}(v)$, where v is the leaf of T_w that stores this entry. This is accessed only during deletions, to efficiently identify the node to be pruned.
- $\underline{EM}$ is an encrypted map mapping node label and access

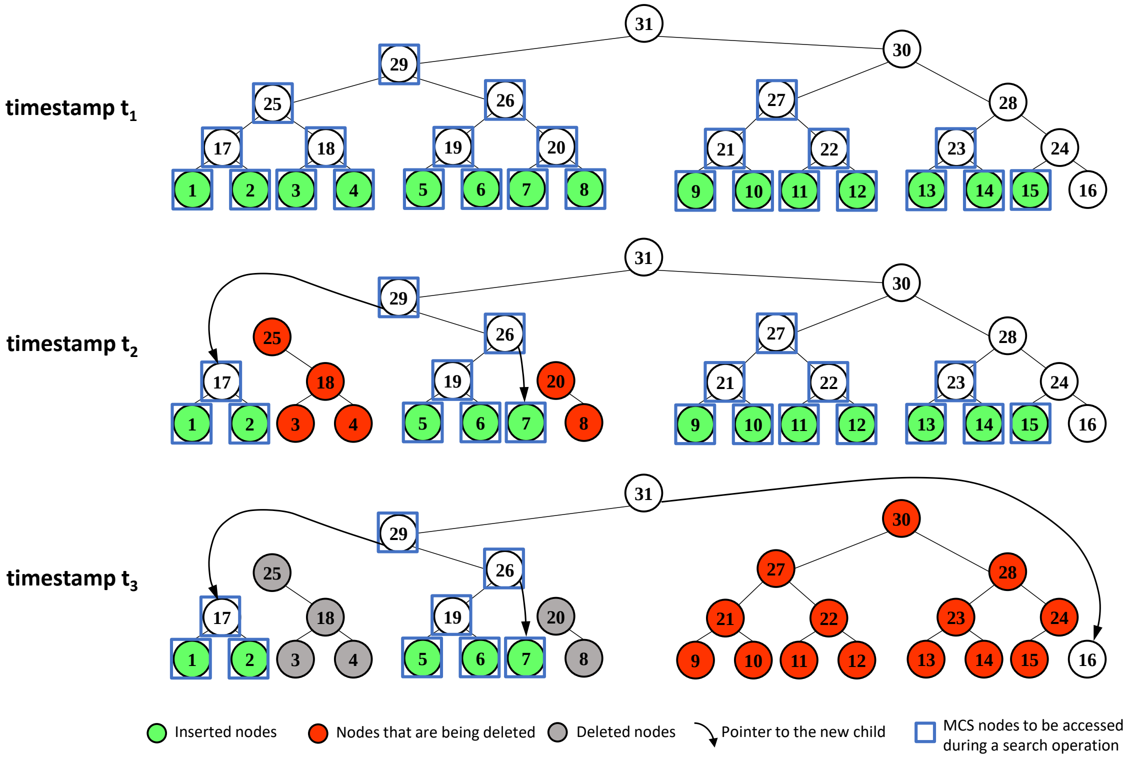


Figure 3: Example of the tree T_w state after executing insertion/deletion with OSSE and LLSE. **(top)** The tree after 15 insertions for w. **(middle)** The tree when deleting node 8 (and consequently its parent 20) and sibling nodes 3,4 (which also deletes their parents 18 and grandparent 25). After these deletions, remaining siblings replace the removed parent nodes (e.g., 7 is now the right chilg of 26). **(bottom)** The tree when deleting nodes 9-15. After the deletion, the only remaining node in the right subtree (node 16) will be the right child of 31 (for future insertion). In all cases, nodes in blue boxes denote the MCS nodes that will have to be accessed in a subsequent search for w.

counter pairs $(\mathsf{lab}(v), acc)$ to: (i) stored document id, if v is a leaf, and (ii) ($v.left$, $v.leftAcc$, $v.right$, $v.rightAcc$), otherwise. It is traversed in searches and modified in updates.

- $\underline{M_{Cnt}}$ is a plaintext map stored at the client that maps keyword w to counters (cnt_w, i_w, d_w), corresponding to number of searches, insertions, and deletions for w.
- $\underline{M_{Tks}}$ is a plaintext map stored at the server that maps pairs of tokens and node labels (tk_0, lab) to tokens ctk. The lookup token tk_0 is unique for each keyword w. This is used to facilitate searches for w that involve nodes that have been written/removed prior to the previous search, and it returns the correct "old" token for this node entry. By default, in each search the server first searches at EM with the latest token tk received from the client, and only in case this returns nothing it falls back to M_{Tks}.

Setup. The client first initializes these data structures (with capacity N for the two OMAPs). It then samples a pseudorandom function key $k \leftarrow PRF.Gen(1^\lambda)$ and a symmetric encryption key $sk \leftarrow SKE.Gen(1^\lambda)$. Its local state σ consists of M_{Cnt} and the state of the two OMAPs, and its key K is (k, sk). Finally, it sends $EDB = \{EM, OM_{Del}, OM_{tree}\}$ to the server. Throughout the scheme, all parties have access to hash functions H_0, H_1 (modelled as random oracles).

Insertion. The process to insert a keyword and document identifier pair (w, id) is shown in Algorithm 1. The client first computes the label of the leaf where this will be stored based on its insertion counter from M_{Cnt} and stores a corresponding entry at OM_{Del}. It also uses the PRF F to compute a token tk that is unique for this keyword and the number of previous searches. Finally, it creates an entry to be stored in EM by the server. It calculates the address by hashing tk, the leaf's label, and its access counter 1, and the value by encrypting id.

Deletion. Removing an entry with OSSE requires more effort, as shown in Algorithm 2. First the client increments the deletions counter d_w and retrieves the label of the node of T_w that stores the entry (w, id) to be removed from OM_{Del} (lines 1-3). The next steps requires retrieving all the nodes of T_w from the root of the tree down to node pos, with a logarithmic number of OMAP accesses. Using the information from the retrieved nodes, the client deduces which are the actual parent pp, grandparent gp, and sibling sib of pos (lines 4-11). Then it can "prune" the tree and remove pos and pp, by setting sib as the direct (left or right) child of gp (lines 12-14).

Finally, it needs to store all this updated information back

Algorithm 1 OSSE.Update$(K,\textbf{add},w,id,\sigma;EDB)$

```
Client:
  $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}.\text{get}(w)$;    $tk \leftarrow F_k(w, cnt_w)$
  $M_{Cnt}.\text{put}(w, (cnt_w, \text{++}i_w, d_w))$
  $OM_{Del}.\text{put}((w,id), i_w)$
  $addr \leftarrow H_0(tk, i_w, 1)$;    $val \leftarrow Enc(sk, (w,id))$
  Send $(addr, val)$ to server
Server:
  $EM.\text{put}(addr, val)$
```

Algorithm 2 OSSE.Update$(K,\textbf{del},w,id,\sigma;EDB)$

```
Client:
 1: $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}.\text{get}(w)$;    $tk \leftarrow F_k(w, cnt_w)$
 2: $M_{Cnt}.\text{put}(w, (cnt_w, i_w, \text{++}d_w))$
 3: $pos \leftarrow OM_{Del}.\text{get}((w,id))$
 4: Let $V, KV \leftarrow \emptyset$
 5: Let Labs be the labels on the path from root to leaf $pos$
 6: for $l \in \text{Labs} \setminus pos$ do
 7:     $v_l \leftarrow OM_{Tree}.\text{get}((w,l))$
 8:     if $v_l$ is null then initialize to default
 9:     Add $v_l$ to $V$
10: Let $pp, gp$ be the parent and grandparent of $pos$
11: Let $sib$ be the label of the sibling of $pos$
12: if $pp$ is left child of $gp$ then
13:     $gp.left \leftarrow sib$;    $gp.leftAcc \leftarrow sib.acc$
14: else $gp.right \leftarrow sib$;    $gp.rightAcc \leftarrow sib.acc$
15: for each $v_l \in V$ do                                ▷ start from root
16:     Let $v_{l'}$ be the child of $v_l$ on the path
17:     if $v_{l'}$ is left child of $v_l$ then $v_l.leftAcc\text{++}$
18:     else $v_l.rightAcc\text{++}$
19:     $v_l.acc\text{++}$;    $OM_{Tree}.\text{put}((w,l), v_l)$
20:     $addr \leftarrow H_0(tk, l, v_l.acc)$;    $t \leftarrow H_1(tk, l, v_l.acc)$
21:     $val \leftarrow t \oplus (v_l.left, v_l.leftAcc, v_l.right, v_l.rightAcc)$
22:     Add $(addr, val)$ to KV
23: Send KV to server
Server:
24: for $(addr, val) \in KV$ do $EM.\text{put}(addr, val)$
```

to EDB. As in insertion, this will take place in two parts. Once in EM for future searches and once in OM_{Tree} for future updates. To guarantee forward privacy, the entries to EM must be unrelated to prior entries for w from the viewpoint of the server. This is done by incrementing the access counter acc of retrieved nodes (stored at all nodes and their parents) and then computing the new EM address-value pairs for these incremented counters (lines 15-23). Addresses are then computed as in insertion. Values are computed by XOR'ing the node's data with the output of H_1 on the same input as for the address. Due to the randomness of H_1 and the pseudorandom way token tk was computed, these writes to EM appear entirely random to the server. We note that some of these nodes are "redundant" as they may have been removed from prior deletions. Still, to hide this information the client puts back their entries, although they will not be used for searches. OM_{Tree} will also store the updated node information (line 19); these accesses do not reveal any information due to the OMAP security.

For en example, see Figure 3(middle). When leaf 8 is removed, its parent 20 is also removed and its sibling 7 will become the new right child of its grandparent 26. Moreover the access counters of all nodes along the path to the root (26, 29, and 31) are increased to 1.

Search. Having spent the necessary effort to maintain and prune T_w during updates, we can now benefit during searches (Algorithm 3). The client computes s, the next-next power of 2 from i_w. This is *the parent of the largest tree of the MCS* and let $start$ be its label. Another way to view it is that this is the highest node in T_w that may be affected due to deletions in the leaf range $[1, i_w]$—if all of them are deleted—which makes it the ideal starting point of the search. (For completeness OSSE maintains an extra node above the root of T_w; we omit it from the presentation for simplicity.) Then, the client needs to calculate the access counter for s. This is very easy due to two facts. First, each deletion increments the access counter of the entire path up to the root of T_w. Second, s lies on this path for all prior deletions for w. Thus, its access counter is d_w which is why we store this deletion counter in M_{Cnt}. Finally, the client sends $(i_w, start, d_w)$ to the server, together with the pseudorandom token tk for this combination of keyword and search counter (and increments this counter), and tk_0, the first search token for w that serves as a unique identifier.

Armed with this information, the server traverses T_w recursively from $start$ to leafs containing the result (ignoring nodes that do not have any descendant leafs in $[1, i_w]$. To "access" a node, it computes its candidate entry in EM, in a way that mimics how this was computed by the client during previous updates. With access to the token tk, the server retrieves the actual information stored for intermediate nodes (i.e., EM is a response-revealing encrypted map): their children and their children's access counters. In this way, it can keep traversing the tree until it either finds leafs, or it hits null entries. Note that, for each label, the server first "tests" the latest token tk (line 13); if this does not return anything, it looks up the entry for tk_0 in M_{Tks} to see if there is an entry to be found for this label from a prior token for w (lines 14-18).

For instance, see Figure 3(bottom). It starts from s (the conceptual node above root 31). Knowing $i_w = 15$, it ignores its right child (node 16), moving to its left child node 27 and from then to 17, 26, etc., each time reading the access counter from the parent. If a leaf is reached, the server adds the retrieved ciphertext from EM to the result (note that leaf access counters are never incremented) (line 19). More interestingly, if a null entry is reached this implies no deletions have taken place among the leafs of the sub-tree rooted at this node. So

Algorithm 3 OSSE.Search$(K, w, \sigma; EDB)$

```
Client:
 1: $(cnt_w, i_w, d_w) \leftarrow M_{Cnt}.\text{get}(w)$;
 2: $tk \leftarrow F_k(w, cnt_w)$;   $tk_0 \leftarrow F_k(w, 0)$
 3: $M_{Cnt}.\text{put}(w, (\text{++}cnt_w, i_w, d_w))$
 4: Let $s$ be the smallest power-of-2 such that $i_w \leq s$
 5: Let $v$ be the root of the subtree of leafs $1, \ldots, 2s$
 6: $start \leftarrow \mathsf{lab}(v)$
 7: Send $(tk, start, i_w, d_w, tk_0)$ to server
Server:
 8: Initialize emptylists TLabs, Res
 9: Append $(start, d_w)$ to TLabs
10: while TLabs $\neq \emptyset$ do
11:     Remove the first entry $(lab, acc)$ from TLabs
12:     if $lab \leq N$ then $acc \leftarrow 1$          ▷ $lab$ is a leaf
13:     $ctk \leftarrow tk$;   $val \leftarrow EM.\text{get}(H_0(ctk, lab, acc))$
14:     if $val$ is null then
15:         $ctk \leftarrow M_{Tks}.\text{get}(tk_0, lab)$
16:         if $ctk \neq$ null then
17:             $val \leftarrow EM.\text{get}(H_0(ctk, lab, acc))$
18:     else $M_{Tks}.\text{put}((tk_0, lab), (tk))$
19:     if $lab \leq N$ then Append $(lab, val)$ to Res  ▷ is a leaf
20:     else if $val \neq$ null then
21:         $data \leftarrow val \oplus H_1(tk, lab, acc)$
22:         Parse $data$ as $(left, leftAcc, right, rightAcc)$
23:         if $left$ or $right$ = $\perp$ then set to default values
24:         Append $(left, leftAcc)$ to TLabs
25:         if $\exists$ leaf $l$ below $right$ with $l \leq i_w$ then
26:             Append $(right, rightAcc)$ to TLabs
27:     else                               ▷ no deletions below $lab$
28:         Let $l$ be the leftmost leaf below $lab$
29:         Let $l'$ be the rightmost leaf below $lab$, with $l' \leq i_w$
30:         Append $(i, 0)$ to TLabs, for $i = l, \ldots, l'$
31: Send Res to client
Client:
32: Initialize set $DB(w) \leftarrow \emptyset$
33: for $(lab, val)$ in Res do Add $Dec(sk, val)$ to $DB(w)$
```

the server can immediately add the EM entries of all these leafs to the result (lines 27-30).

4.3 OSSE Efficiency & Security Analysis

We now analyse OSSE in terms of its asymptotic efficiency and its achieved security properties. First, in Appendix C.1 we prove the following lemma regarding its search efficiency.

Lemma 1. *For a keyword w for which i_w insertions have taken place and $|DB(w)| = n_w$, the search process accesses at most $2n_w + \log i_w$ nodes from T_w.*

From the above, it follows that search takes $O(n_w + \log i_w)$ operations, i.e., linear in the result size and independent from the number of insertions. Moreover, searches require a single roundtrip to complete. Regarding updates, insertions take just one oblivious map and encrypted map access. Deletions require $O(\log N)$ such operations to access the path above the node to be removed. The setup takes $O(N)$ time (for N an upper bound on the number of entries in the dataset). Finally, client storage consists of three counters per keyword in $|W|$, as well as the oblivious maps' client state and the secret keys. The server storage is linear in the total number of entries N but it grows logarithmically with the number of deletions D, so it is overall $O(N + D\log N)$. In the absolute worst case, if all entries are deleted, this becomes $O(N \log N)$.

The correctness of OSSE (i.e., that a search returns the correct result $DB(w)$, as formulated by all prior updates for w) follows from the construction and the correctness of the underlying primitives, with one exception. If two distinct inputs to H_0 or H_1 (or the same input for the two different hash functions) map to the same hash value, this affects the correctness of the scheme, e.g., as the latter entry will "overwrite" the prior one. This is a common issue with prior DSE works that use an encrypted map as we do (e.g., [8, 27]) and the solution is to pick the range of H_0, H_1 so as to make this probability negligible in the security parameter λ. Let M be the total number of entries that will be stored in EM throughout the execution (for OSSE, $M \leq N \log N$). If we then set the range of H_0, H_1 to $2^{\lambda + 2\log M}$, a simple birthday problem analysis guarantees the negligible probability of this event.

Regarding the security of OSSE, we can analyze its leakage as follows. First, during setup the server only learns the upper bound N. During updates, note that the server's view consists of two types of data: (i) the messages involved in the oblivious map access (or accesses for deletions), (ii) the $(addr, val)$ pairs it inserts to the encrypted map. For (i), assuming a secure oblivious map these leak no information. For (ii), each of these is computed with a new keyword/token/access-counter combination, even when it pertains to a previously accessed node, using a secure PRF. Without access to the token tk used as input to this PRF (this will only be revealed during a subsequent search), these addresses and values are indistinguishable from random when they refer to intermediate nodes. When pertaining to leafs, the values are encrypted using a secure encryption scheme, and exactly one such value is sent for insertions. Finally, the total number of accesses and entries is fixed depending only on the type of operation (and N for deletions). From the above, OSSE satisfies forward privacy.

In a search for keyword w, the server learns tokens tk_0, tk. The first reveals when previous searches for w took place. With the second, the server can exhaustively access EM for all labels and learn the entire topology of T_w. Remembering when these entries were written reveals when prior insertions and deletions for w took place. Moreover, by extracting all versions of T_w nodes for different access counters and correlating this with the update timestamps reveals the specific insertion that each deletion cancelled. However, since document ids remain encrypted, the server cannot learn which documents contained w but have since been removed (if not search for w took place since). Based on this leakage profile, OSSE satisfies backward privacy. We formally state and prove its security in Appendix C.2.

5 Log-Log Search DSE with Improved Storage & Deletion (LLSE)

Our OSSE scheme achieves excellent search performance, however, during deletions it performs a logarithmic number of oblivious/encrypted map accesses. Moreover, this also increases storage. Here we present our second scheme, LLSE, that requires only $O(1)$ map accesses during updates and has $O(N)$ storage, at the trade-off of an extra $O(\log\log i_w)$ factor for searches (which is still asymptotically better than the previous state-of-the-art quasi-optimal search scheme from [19].

The main idea behind LLSE is to still use the same T_w tree structure but enhance it so that each node has a parent pointer. This allows us, during deletes, to access directly the node to be removed, its parent, and its grandparent, without starting each time from the root. This simple idea introduces a negative side-effect: during searches we can no longer rely on every node to store its children's access counters. To overcome this, for each encountered node, the server performs a binary search to find the largest acc for which there exists an entry in EM. A careful analysis shows that the upper bound for this binary search is roughly twice the height of the node in T_w, which yields the $O(\log\log i_w)$ extra overhead. The actual implementation of this is trickier as one needs to maintain access counters across searches (without re-encrypting every node/access-counter combination entry, as this increases the search overhead to $O(\log i_w)$), as we explain in detail next.

5.1 LLSE Data Structures & Algorithms

First we describe the involved data structures. These are the same as for OSSE, with the following exceptions.

- $\underline{OM_{Tree}}$ stores for node v its own access counter acc, and its left child, right child, and parent ($v.left$, $v.right$, $v.prnt$), but not their access counters (see Figure 2).
- $\underline{EM}$ is the same as OSSE, except for non-leaf nodes v for which it stores ($v.left$, $v.right$), but not access counters.
- $\underline{M_{Cnt}}$, instead of d_w, stores locally the label of the correct parent for the next future insertion. Initially, this is $N+1$ (the natural parent of the leaf with label 1). After i_w insertions, it is set to the natural parent of the leaf with label i_w+1. However, if the most recent updates for w are a series of one or more deletions, the parent for the next insertion lies higher in T_w. The correct value for this pointer is calculated by the client during updates.
- $\underline{M_{Tks}}$ now stores, not only the prior token version for a label, but also the access counter of the latest previous entry. The goal of this is subtle but crucial: It is necessary for launching the binary search for the correct range. This is best illustrated by an example. Consider that during the first search for w, the binary search for node v with height h is correctly executed in the range $[0,2h]$ and the result is acc. In the second search for w the server will try to run the same binary search in vain as at least some of the entries are the ones that were calculated with the old token (from the first search)! To avoid this, M_{Tks} stores for v the old token and the access counter acc, so that the server can retrieve it and then launch the next binary search in $[acc,2h]$.

Algorithm 4 LLSE.Update$(K,op,w,id,\sigma;EDB)$

```
Client:
 1: $(cnt_w, i_w, nxtPrnt_w) \leftarrow M_{Cnt}.\text{get}(w)$;   $tk \leftarrow F_k(w, cnt_w)$
 If update is insertion $(op = add)$
 2: $OM_{Del}.\text{put}((w,id), i_w)$
 3: $OM_{Tree}.\text{put}((w,i_w),(\perp,\perp,nxtPrnt_w))$
 4: $nxtPrnt_w \leftarrow$ label of the parent of $i_w+1$ (unused) leaf
 5: $M_{Cnt}.\text{put}(w,(cnt_w, ++i_w, nxtPrnt_w))$
 6: $addr \leftarrow H_0(tk, i_w, 0)$;   $val \leftarrow Enc(sk,(w,id))$
 7: Send $(addr, val)$ to server
 If update is deletion $(op = del)$
 8: $pos \leftarrow OM_{Del}.\text{get}((w,id))$
 9: $v \leftarrow OM_{Tree}.\text{get}((w,pos))$
10: $pp \leftarrow OM_{Tree}.\text{get}((w,v.prnt))$            ▷ parent node
11: $gp \leftarrow OM_{Tree}.\text{get}((w,pp.prnt))$      ▷ grandparent node
              ▷ if parent/grandparent not found set it to default
12: if $v$ is left child of $pp$ then $sibPos \leftarrow pp.right$
13: else $sibPos \leftarrow pp.left$
14: $sib \leftarrow OM_{Tree}.\text{get}((w,sibPos))$            ▷ sibling node
15: $sib.prnt \leftarrow pp.prnt$
16: if $pp$ is left child of $gp$ then $gp.left \leftarrow sibPos$
17: else $gp.right \leftarrow sibPos$
18: $gp.acc{+}{+}$
19: Update $nxtPrnt_w$ if it is affected
20: $M_{Cnt}.\text{put}(w,(cnt_w, i_w, nxtPrnt_w))$
21: $addr \leftarrow H_0(tk, pp.prnt, gp.acc)$
22: $val \leftarrow H_1(tk, pp.prnt, gp.acc) \oplus (gp.left, gp.right)$
23: $OM_{Tree}.\text{put}((w,sibPos), sib)$
24: $OM_{Tree}.\text{put}((w,pp.prnt), gp)$
25: Send $(addr, val)$ to server
Server:
26: $EM.\text{put}(addr, val)$
```

Setup. As in OSSE, this process initializes all data structures and generates keys. The one difference is the capacity of OM_{Tree} which is $2N$ as it will store all nodes, including leafs.

Insertion. This proceeds very similarly with OSSE, as shown in Algorithm 4. The main difference is that it takes one additional OMAP access in order to write the new leaf's information to OM_{Tree} (line 3). This is necessary for finding its parent quickly, in case it is deleted in the future.

Deletion. This is basically a "lightweight" version of deletions with our first scheme. Since nodes store their parents' labels, we can now retrieve the node *pos* to be deleted from in OM_{Tree}, and directly from this its parent and grandparent (Algorithm 4, lines 9-11), without having to fetch the entire path from the root. The remaining of the algorithm is essentially the same as before, except for the need to calculate what the next parent's label *nxtPrnt* for w will be. This is only affected if the current deletion removes the latest entry, or is part of a series of consecutive deletions that includes the latest entry. Finally, this process adds only one entry to EM, for the updated grandparent node, which makes the storage grow only linearly with the total number of operations.

Search. The search process for LLSE is shown in Algorithm 5. The client first sends the necessary data to bootstrap the search. The server traverses the tree from *start* in largely the same manner as in OSSE, with one crucial difference: it does not know the correct *acc* for the nodes it encounters. This is resolved as follows. First (lines 12-15), it checks whether there is a previous entry in M_{Tks} for this keyword/label combination. If found, then it extracts *sacc*, the access counter that node *lab* had when (and if) it was last accessed *during* the previous search for w, and *otk* which is the token during that search. Clearly, if the node has even been written, the correct current *acc* can only be larger or equal than *sacc*. In both cases, it then begins a binary search in the range between this value (or 0, if not found) and twice the *lab* node's height, looking for the largest value *acc* for which the entry for *lab* in EM is not empty, which is indeed the correct *acc* for node *lab*. When this counter is computed, the correct data for *lab* is retrieved from EM with the appropriate token (lines 21-25). The search then progresses to the retrieved node's children, or its "natural" children if no entry is found for *lab* (lines 26-31).

5.2 LLSE Efficiency & Security Analysis

The search efficiency of LLSE is similar to OSSE. Each search entails the same tree traversal, so it accesses $O(n_w + \log i_w)$ nodes. But this time retrieving each node's data requires a binary search with max range $[0, 2h]$, where $h \leq \log N$ is the node's height. This follows from the fact that for a node at height h each of its children pointers can change due to deletions at most $h-1$ times (each time moving "up" a layer in the tree). Therefore, with LLSE searches require $O((n_w + \log i_w) \cdot \log\log N)$ operations. Updates require $O(1)$ oblivious and encrypted map accesses, both for insertions and for deletions—an $O(\log N)$ factor faster than OSSE for the latter. Finally, the storage is linear in both insertions N and deletions D; since $D \leq N$, its overall storage is optimal $O(N)$. Other efficiency metrics are asymptotically the same as OSSE. It is easy to see the two schemes have the same leakage profile; a curious server can ignore the binary search and try all access counters to extract all T_w node information from EM. Thus, LLSE is also forward-and-backward private.

Algorithm 5 LLSE.Search$(K, w, \sigma; EDB)$

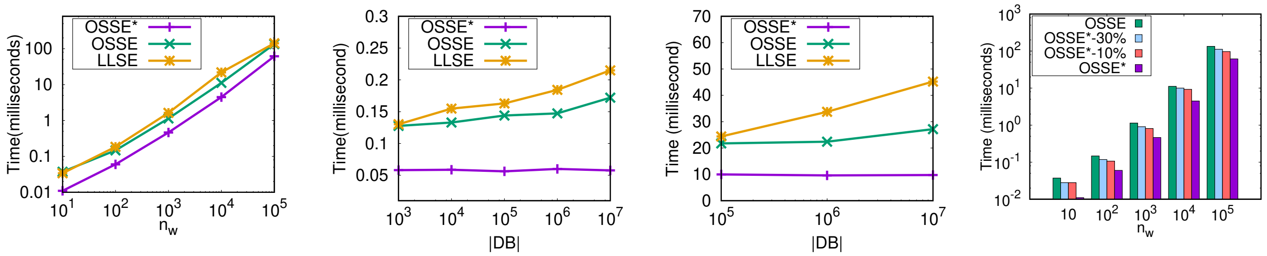
```
Client:
 1: $(cnt_w, i_w, nxtPrnt_w) \leftarrow M_{Cnt}$.get($w$);
 2: $tk \leftarrow F_k(w, cnt_w)$;    $tk_0 \leftarrow F_k(w, 0)$
 3: $M_{Cnt}$.put($w$, $(\text{++}cnt_w, i_w, d_w)$)
 4: Let $s$ be the smallest power-of-2 such that $i_w \leq s$
 5: Let $v$ be the root of the subtree of leafs $1, \ldots, 2s$
 6: $start \leftarrow \mathsf{lab}(v)$
 7: Send $(tk, start, i_w, tk_0)$ to server
Server:
 8: Initialize emptylists TLabs, Res
 9: Append $start$ to TLabs
10: while TLabs $\neq \emptyset$ do
11:     Remove the first entry $lab$ from TLabs
12:     $sacc \leftarrow 0$;    $ctk \leftarrow tk$
13:     $r \leftarrow M_{Tks}$.get$(tk_0, lab)$
14:     if $r \neq$ null then parse $r$ as $(otk, osacc)$
15:         $sacc \leftarrow osacc$;    $ctk \leftarrow otk$
16:     if $lab \leq N$ then                         ▷ $lab$ is a leaf
17:         $val \leftarrow EM$.get$(H_0(ctk, lab, 1))$
18:         Append $(lab, val)$ to Res;    continue
19:     Let $h$ be the height of node $lab$
20:     Perform binary search in $[sacc, 2h]$ to find max $acc$
          such that $EM$.get$(H_0(tk, lab, acc)) \neq$ null
21:     if $acc > sacc$ then
22:         $val \leftarrow EM$.get$(H_0(tk, lab, acc))$
23:         $M_{tks}$.put$((tk_0, lab)), (tk, acc))$
24:     else if $acc = sacc$ and $acc > 0$ then
25:         $val \leftarrow EM$.get$(H_0(ctk, lab, acc))$
26:     if $acc > 0$ then
27:         $(left, right) \leftarrow val \oplus H_1(tk, lab, acc)$
28:     else $(left, right) \leftarrow$ natural children of $lab$
29:     Append $left$ to TLabs
30:     if $\exists$ leaf below $right$ with label $\leq i_w$ then
31:         Append $right$ to TLabs
32: Send Res to client
Client:
33: Initialize set $DB(w) \leftarrow \emptyset$
34: for $(lab, val)$ in Res do Add $Dec(sk, val)$ to $DB(w)$
```

We formally state its security in Appendix C.2; due to lack of space, we delegate the proof to the full version.

6 Optimizations

In this section, we present some optimizations that can improve the performance of our schemes.

(I) Improving Tree Traversals (during search). The search algorithms of OSSE and LLSE need to search the sub-trees that contain the MCS nodes for the range $[1, i_w]$. We can optimize this by storing as additional metadata two bits for each



(a) (b) (c) (d)

Figure 4: Search computation time vs. (a) variable n_w for $|DB| = 1$M, (b) variable $|DB|$ for $n_w = 100$, (c) variable $|DB|$ for $n_w = 20K$, (d) variable n_w for $|DB| = 1$M and different OSSE cache settings.

MCS root node, which are used to: (i) avoid deleted sub-trees and (ii) start the search from the parents of the MCS nodes that contain a single leaf. To maintain these metadata bits, during deletions we simply check whether the leaf to be deleted is the last or second-last in its MCS sub-tree. Otherwise, the metadata do not change. During search, these bits are sent to the server who then chooses its best strategy. For example, in Figure 3(bottom), only MCS root node 29 is searched, because all the other MCS sub-trees are empty. In cases like this, where almost all MCS sub-trees are empty, this optimization can reduce the number of operations to linear in n_w. It also works naturally with search result caching, which we describe next. Finally, this does not affect the leakage of the scheme (a "curious" server can already access all tree nodes anyway).

(II) Search Result Caching. Similar to [16], we propose a second optimization in which previous results can be cached at the server. This entirely avoids sub-tree traversals for unchanged nodes (since the last search) and directly returns the cached results. This is done by using a simple hashmap at the server and storing the returned results in this map using the search token as the key. The client sends the previous search token together with the next search query, and the server uses it to find cached results (if any). Identifying whether a node has been changed between two consecutive searches can be done using its access counter in OSSE (and, potentially, in LLSE at the cost of additional bookkeeping). For example, in Figure 3 assume a search query was performed before the deletion of nodes 9-15 and a second query is performed after the deletion of these nodes (in Figure 3(bottom)). In this case, the second query will get the results for node 29 directly from the cache (since that sub-tree remains unchanged). We note this optimization does not incur extra leakage as the information of when past updates and searches for w took place is already part of our schemes' search leakage profile.

(III) Storage Clean-Up in OSSE. When deleting a leaf in OSSE, $\log N$ nodes are re-written with incremented access counters. This is necessary to avoid leaking information about the tree topology, but it also causes an extra logarithmic write and space overhead.We can use a similar approach to Java's background garbage collection, so that the server will be able to remove "stale " records that are identified *after* a relevant search query (exposing stale records during deletes would violate forward privacy definition). This can run periodically or in the background between queries to keep storage low. We experimentally evaluate its impact in Section 7.3.

(IV) Constant Client Storage. Our schemes require non-constant client storage to maintain the necessary metadata M_{Cnt}, M_{Tks}. An approach to reduce the DSE client storage that has already been proposed in [19] and is applicable in our schemes is to store M_{Cnt}, M_{Tks} at the server using OMAPs, at the cost of additional roundtrips of communication. Due to the oblivious property, all these accesses are simulatable, revealing no additional information to the server. In practice this mainly affects the search performance since it takes an OMAP access to retrieve the metadata for w—an additive $O(\log^2 N)$ factor. We experimentally evaluate this approach for our schemes in Section 7.2.

7 Experimental Evaluation

We now report on the performance of our schemes and compare them with prior state-of-the-art DSE. We implemented OSSE and LLSE with approximately $10K$ lines of code in C++. Our code is available at https://github.com/jgharehchamani/OS-SSE. For symmetric encryption, PRF, and hashing we used the AES implementation of OpenSSL [2]. For comparison with Horus, QOS, and SD$_d$ we used the publicly available code of [13, 19], also in C++. We used a machine with eight-core Intel Xeon E-2174G 3.8GHz processor, 128GB RAM, 1TB SSD, running Ubuntu16.04 LTS. In all our experiments, the database is stored in RAM.

We focus on measuring computation time and communication size for Search and Update queries. We measure these for variable size synthetic datasets with $|DB| = 10^3$-10^7 records, each time setting the total number of distinct keywords $|W|$ to one-hundredth of $|DB|$. Likewise, we report results for varying result size n_w between 10-10^5 documents. Moreover, to demonstrate the impact of deletions in the search performance we report result for variable deletion rates from 0-90%. Finally, since our OSSE scheme is the first to achieve optimal search but it has asymptotically increased server storage, we run a specially designed experiment to demonstrate that in practice this storage blowup is often not prohibitively large. Experiments were repeated 10× and the average is reported.

7.1 Performance of our Schemes

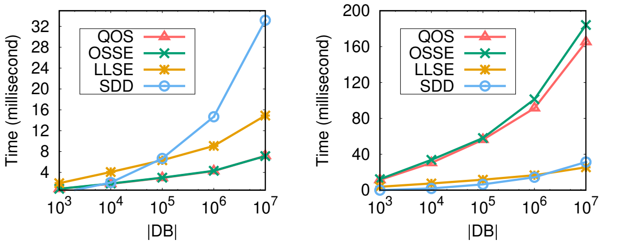
Our first set of experiments focuses on the performance of OSSE, LLSE, and on evaluating the effect of our result-caching optimization from Section 6. Specifically, OSSE* denotes the version of OSSE with result caching. Regarding smart tree traversal, we integrated this optimization to both OSSE and OSSE* and we found it to have very small impact in most cases. For this part, we fixed the deletion rate to 10%.

Search Performance. Figure 4 shows search computation time as the result size (a) and database size (b,c) change. First, as expected from our analysis search time grows linearly with n_w whereas its change with respect to the database size $|DB|$ is not considerable (note the exponential growth in the x-axis). More importantly, all our schemes have excellent performance in practice. E.g., even for retrieving large results of $100K$ id's from a dataset of size $1M$ they take less than 200ms, whereas for smaller result sizes their performance is even better!

In Figures 4(a-c), OSSE* specifically refers to "best-case" result caching, i.e., when two searches for w take place without an intermediate update, so the second search can maximally benefit from caching. Not surprisingly, OSSE* outperforms the other two schemes by 2.1-4.8×. To better evaluate the effect of result caching on OSSE, we ran an experiment with variable caching benefit, shown in Figure 4(d). In this setting, after inserting a number of entries for w and removing 10%, we execute a search for w to fill the cache. We then delete additional entries so as to invalidate a certain percentage of the cache for w and report the time for a second search and variable result size for: (i) OSSE* where the entire result is in cache, (ii) OSSE* with 10% and (iii) OSSE* with 30% of the cached result is modified, and (iv) OSSE-none of the cash can be re-used. As expected, the more modified the result is the second time, the smaller the improvement achieved. E.g., for $|DB| = 1M$ and result size $100K$ OSSE* takes 61ms, OSSE* with 10% and 30% modification takes 97ms and 111ms, and OSSE takes 134ms. In that sense, OSSE and OSSE* represent an upper and lower bound for the search performance of our first scheme, and in the following experiments for search performance we report both of these.

Finally, search communication size is the same for both OSSE and LLSE and it is concretely *optimal*. The client sends at most two search tokens and the server responds with *exactly* n_w ciphertexts. For example, for $|DB| = 1M$, result size $1K$ and 128-bit AES ciphertexts this is approximately 16KB.

Update Performance. Figure 5 shows the update computation time for OSSE and LLSE vs. variable $|DB|$. The obvious conclusions from the figures are: (i) the update time slightly increases with DB due to the increase in the size of the OMAPs (again, note the exponential x-axis), (ii) OSSE has faster insertion than LLSE as it needs one oblivious map access while LLSE needs two, (iii) as expected, deletion with LLSE is much faster than OSSE. Overall, OSSE is 2-2.2× faster than LLSE in insertion and LLSE is 3.2-7.2× faster than OSSE in deletion. That said, both schemes have good performance, e.g., OSSE and LLSE take 4ms and 9ms to insert an entry and 101ms and 16ms to delete one from a database of size 1M. Update communication sizes follow the same trend: larger in LLSE for insertion, but smaller for deletion.



(a) (b)

Figure 5: (a) Insertion, (b) Deletion computation time vs. $|DB|$

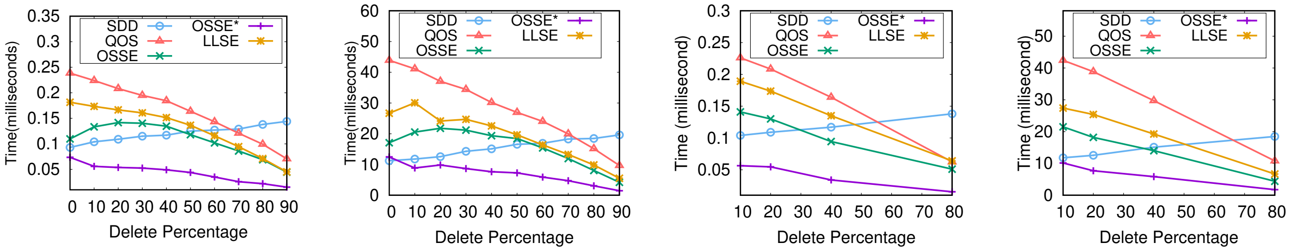
7.2 Comparison with Other DSE

Next, we compare the performance of our schemes with prior forward and backward private schemes, focusing on existing DSE with quasi-optimal search, QOS from [19] and Horus from [13]. We also compare their performance with SD$_d$ from [19], a state-of-the-art DSE that achieves very fast search but linear in the total number of insertions a_w.

Search Performance. Our main motivation for studying optimal DSE schemes is to avoid "paying" for deleted entries during a search. Thus, we ran a set of experiments with variable deletion percentages to measure the effect of deletions for the different schemes. We also included Horus in our experiments, however, it is not shown in the figures, as it was 2-5 orders of magnitude slower than all other schemes.

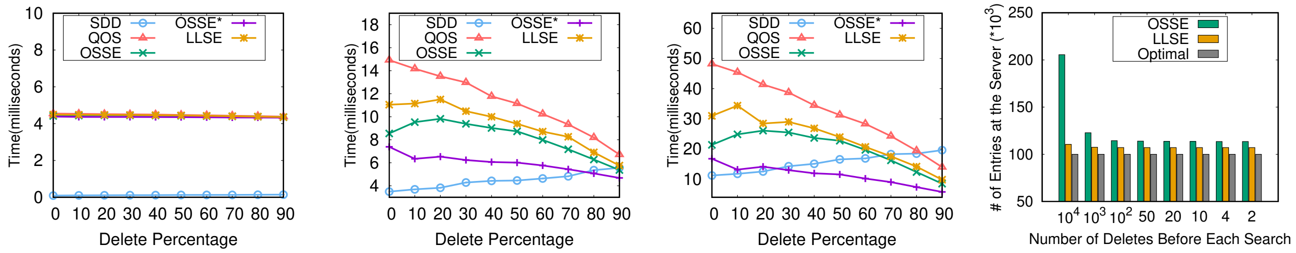
Random deletions. First we vary the deletion percentage between 0-90% after a fixed number of insertions for the queried keyword, with deletions chosen *at random* among them. We try two cases: (a) small results $i_w = 100$, (b) big results $i_w = 20K$, in Figures 6 (a),(b), respectively. The search time of all optimal/quasi-optimal schemes generally decreases with deletion percentage increases (except for OSSE that for 0-20% slightly increases as it is already optimal initially), while for SD$_d$ it grows as expected, since it stores deletions as entries. Second, our schemes OSSE,LLSE, OSSE* outperform the prior best quasi-optimal search DSE QOS across all delete percentages and are 1.4-2.6×, 1.2-1.8×, 3.2-6.6× faster, respectively. The encouraging side-effect of this is that they become truly faster than SD$_d$ at 55%, 55%, and 5% deletion rates for small results and at 45%, 50%, and 5% deletion rates for large results, respectively. In contrast, QOS becomes faster than SD$_d$ only after 65% and 75% deletion for the two settings. This is the joint effect of our compact tree design, elimination of re-encryption in searches, and optimizations. In practice, we believe this really drives the case for the practicality of DSE with search time independent of past deletions.

Structured deletions. We repeated this comparison for the case of more "structured" deletions in which consecutive tree



(a) (b) (c) (d)

Figure 6: Search computation time for $|DB| = 1$M and (a) variable deletion percentage for $i_w = 100$, (b) variable deletion percentage for $i_w = 20$K, (c) structured deletion for $i_w = 100$, (d) structured deletion for $i_w = 20$K.



(a) (b) (c) (d)

Figure 7: Search computation time for $|DB| = 1$M and variable deletion percentage for: (a) $i_w = 100$ using OMAP, (b) $i_w = 5$K using OMAP, (c) $i_w = 20$K using OMAP. (d) Number of entries in server for $i_w = 100$K and 30K deletion and variable search.

leafs are pruned in batch to see how this affects performance. In practice, this deletion pattern may occur in applications such as an encrypted relational database that performs a bulk update in a table, effectively removing all entries for a column and inserting new ones. Subsequent such modifications will again delete consecutively place entries. To evaluate this, we insert a fixed number of entries for w and then delete a variable percentage of them by choosing consecutive ranges of size $1, 2, 4, 8$ at random with 10% probability. In this manner, we achieve deletion rates of $10, 20, 40, 80\%$ but in a certain structure. The result is shown in Figure 6 (c),(d). The general observation is that the impact of this is not significant, with all schemes achieving slightly better performance (roughly 1.1-1.7×) as these deletions eliminate entire sub-trees and reduce the number of accessed nodes in a "nice" manner, but the improvement is not that considerable. The one exception to this is SD$_d$ that is unaffected as it treats all deletions equally.

Small-client storage versions. Demertzis et al. [19] consider a scenario with small-client users who cannot store keyword counters locally. For QOS, they do this by storing them in an oblivious map of size $|W|$ and retrieving them at the beginning of each search. As discussed in Section 6, we can do the same for our schemes. One expects that this OMAP access will dominate the search time for all schemes. Indeed this is shown for small results in Figure 7(a) where all schemes except SD$_d$ have essentially the same performance. SD$_d$ outperforms all other candidates as it achieves small client storage without the need for an oblivious map in search. However, for medium ($i_w = 5K$) and larger ($i_w = 20K$) result sizes (Figure 7(b),(c)), surprisingly our schemes eventually outperform *all competitors*. E.g., for large result size 20K, OSSE*, OSSE, LLSE, are always better than QOS and even outperform SD$_d$ at 25%, 65%, and 70%, deletion rates, respectively.

Update performance. Regarding insertion computation times (Figure 5), OSSE and QOS have almost the same insertion (and about 2× faster than LLSE). For deletions, LLSE outperforms QOS by 3-6.4× faster than QOS.

7.3 Effect of Deletion Storage for OSSE

One of the drawbacks of OSSE is that it needs to store $\log N$ entries in the encrypted map for every deletion. In the absolute worse case (number of deletions $D = N$), this blows the storage for *EM* by a logarithmic factor. That said, there are two important observations that point towards this not being such a prohibitive factor. First, the server's storage also consist of two OMAPs that are setup to max capacity N and their size is not affected by deletions, so the blowup is only partial. Second, as we explained in Section 6, the server can perform a "clean-up" in the background or periodically, using the tokens it gathers from searches to remove unnecessary "stale" entries. Here, we attempt to demonstrate the effect this cleanup can have in keeping the storage smaller. The crucial measure here is how often searches take place, i.e., the relative frequency of searches and deletions for a keyword.

In lack of a real-world query workload that would allow us to accurately evaluate this, we perform a simplified experiment where we first insert 100K entries for a keyword and then we randomly delete 30%. What we vary is how searches are interleaved with these 30K deletions among $\{1/10K, 1/1K, 1/100, 1/50, 1/20, 1/10, 1/4, 1/2\}$. E.g., 1/2 means that each deletion is followed by a search for the keyword, whereas $1/10K$ means only three searches happen, each after $10K$ deletions. What we report in Figure 7(d) is the average number of entries in *EM* throughout the execution of all these searches/deletions (assuming for simplicity that cleanup happens immediately after each search). For comparison we also include the same number for LLSE, as well as an "optimal" plaintext storage where deletions just mark a cell as available.

As expected, lower search frequency leads to overall increase in the server's storage in OSSE. However, even when search occurs after 10K deletions the overall average storage is at most 2× the optimal size 100K, as a result of cleanup. Besides the average, we also measured the 75-th and 90-th percentile as 280K and 116K entries, respectively. This can be interpreted as 90% of the time the entries' storage for this keyword is less than 3× the optimal and 75% of the time it is less than 1.16×. While our simplified experiment runs for one w, if we assume similar query distribution across keywords these results can be extrapolated for the dataset. On the other hand, with LLSE the storage is almost the same for different search intervals, e.g., for our experiments the average is 1.07-1.1× the optimal number of entries.

8 Conclusion

In this work, we presented two forward-and-backward private DSE schemes with very efficient search performance, independent of prior deletions. The experimental evaluation shows our schemes outperform prior ones for a variety of settings. Our results leave several possible directions for improvement, e.g., propose schemes with similar search performance but faster updates, or DSE that achieve stronger variants of backward privacy (without compromising search efficiency). Finally, motivated by leakage abuse attacks [7,10,32,33,36,46,52,66], it seems very promising to try and adopt leakage-suppression techniques [21,40,54,67] to combine optimal search performance with improved leakage profiles.

References

[1] Mongodb. http://www.mongodb.com/.

[2] OpenSSL: The open source toolkit for SSL/TLS. https://www.openssl.org/.

[3] Pixek: An App That Encrypts Your Photos From Camera to Cloud. https://www.wired.com/story/pixek-app-encrypts-photos-from-camera-to-cloud/.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8:1792–1803, 2015.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. In *EUROCRYPT 2020*, pages 403–432, 2020.

[6] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *STOC 2016*, page 1101–1114, 2016.

[7] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. *NDSS*, 2020.

[8] Raphael Bost. Σοφος: Forward Secure Searchable Encryption. In *ACM CCS*, 2016.

[9] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[11] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*, 2013.

[12] David Cash, Ruth Ng, and Adam Rivkin. Improved Structured Encryption for SQL Databases via Hybrid Indexing. In *ACNS 2021*, pages 480–510, 2021.

[13] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *ACM CCS*, 2018.

[14] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS*, 2005.

[15] Melissa Chase and Seny Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.

[16] Tianyang Chen, Peng Xu, Wei Wang, Yubo Zheng, Willy Susilo, and Hai Jin. Bestie: Very practical searchable encryption with forward and backward security. In *ESORICS 2021*, pages 3–23, 2021.

[17] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS*, 2006.

[18] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[19] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[20] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. *CRYPTO*, 2018.

[21] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security 2020*, pages 2433–2450, 2020.

[22] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *TODS*, 2018.

[23] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. Practical private range search revisited. In *SIGMOD*, 2016.

[24] Ioannis Demertzis and Charalampos Papamanthou. Fast Searchable Encryption With Tunable Locality. In *SIGMOD*, 2017.

[25] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. Efficient searchable encryption through compression. *PVLDB*, 2018.

[26] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[27] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PETS*, 2018.

[28] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[29] Facebook. Myrocks. http://myrocks.io/.

[30] Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In *AsiaCCS 2021*, pages 516–525, 2021.

[31] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.

[32] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenny Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range series. In *ACM CCS*, 2018.

[33] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *ACM CCS 2019*, pages 361–378, 2019.

[34] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In *PKC*, 2018.

[35] F Hueske. State TTL for Apache Flink: How to Limit the Lifetime of State. https://www.ververica.com/blog/, 2018.

[36] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *CODASPY*, pages 235–246. ACM, 2014.

[37] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*, 2017.

[38] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptol. ePrint Arch.*, 2018:978, 2018.

[39] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In *ASIACRYPT*, 2018.

[40] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.

[41] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[42] Seny Kamara, Tarik Moataz, Andrew Park, and Lucy Qin. A decentralized and encrypted national gun registry. *IEEE S&P*, 2021.

[43] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, 2013.

[44] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.

[45] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, 2017.

[46] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE S&P*, 2018.

[47] Russell W. F. Lai and Sherman S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, 2017.

[48] Kasper Green Larsen and Jesper Buus Nielsen. Yes, There is an Oblivious RAM Lower Bound! In *CRYPTO 2018*, pages 523–542, 2018.

[49] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *CCS*, 2015.

[50] Tatsuya Midorikawa, Akihiro Tachikawa, and Akira Kanaoka. Helping Johnny to Search: Encrypted Search on Webmail System. In *AsiaJCIS*, 2018.

[51] Ian Miers and Payman Mohassel. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In *NDSS*, 2017.

[52] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, 2021.

[53] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Lower bounds for encrypted multi-maps and searchable encryption in the leakage cell probe model. In *CRYPTO 2020*, pages 433–463, 2020.

[54] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM CCS*, pages 79–93, 2019.

[55] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *PVLDB*, 12(11):1664–1678, 2019.

[56] Cédric Van Rompay, Refik Molva, and Melek Önen. Multi-user searchable encryption in the cloud. In *ISC 2015*, 2015.

[57] Cedric Van Rompay, Refik Molva, and Melek Onen. Secure and scalable multi-user searchable encryption. In *SCC Workshop*, 2018.

[58] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. Lethe: A tunable delete-aware lsm engine. In *SIGMOD*, 2020.

[59] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.

[60] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[61] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

[62] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph Liu, Surya Nepal, and Dawu Gu. Practical non-interactive searchable encryption with forward and backward privacy. In *NDSS*, 2021.

[63] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[64] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption with optimal search time. In *AsiaCCS 2021*, pages 252–264, 2021.

[65] J. Yang, Z. Liu, J. Li, C. Jia, and B. Cui. Multi-key searchable encryption without random oracle. In *INCoS 2014*, pages 79–84, 2014.

[66] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of File-Injection attacks on searchable encryption. In *USENIX Security 2016*, pages 707–720, 2016.

[67] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. Volume-hiding dynamic searchable symmetric encryption with forward and backward privacy. Cryptology ePrint Archive, Report 2021/786, 2021.

A Details of Cryptographic Primitives

Pseudorandom functions. Let $Gen(1^{\lambda}) \in \{0,1\}^{\lambda}$ be a key generation function, and $F : \{0,1\}^{\lambda} \times \{0,1\}^{\ell} \rightarrow \{0,1\}^{\ell'}$ be a pseudorandom function (PRF) family. $F_k(x)$ denotes $F(k,x)$. F is a secure PRF family if for all PPT adversaries Adv, $|\Pr[k \leftarrow Gen(1^{\lambda}); \mathsf{Adv}^{F_k(\cdot)}(1^{\lambda}) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)}(1^{\lambda}) = 1]| \leq v(\lambda)$, where $R : \{0,1\}^{\ell} \rightarrow \{0,1\}^{\ell'}$ is a truly random function.

Symmetric-Key Encryption. A symmetric-key encryption scheme $SKE = (Keygen, Enc(sk,x), Dec(sk,c))$ consists of the three algorithms:

- $KeyGen(\lambda)$: A probabilistic algorithm that takes the security parameter λ as input and outputs a secret-key sk.
- $Enc(sk,x)$: A probabilistic algorithm that takes sk as input key and a plaintext x. It outputs a ciphertext c.
- $Dec(sk,c)$: A deterministic algorithm that takes as input key sk and ciphertext c. It outputs decrypted plaintext x.

```
$b \leftarrow \mathsf{Real}^{\mathsf{DSE}}_{\mathsf{Adv}}(\lambda, q)$:
$N \leftarrow \mathsf{Adv}(1^\lambda)$
$(K, \sigma_0, EDB_0) \leftarrow$Setup$(1^\lambda, N)$
for $k = 1$ to $q$ do
    $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$
    if $type_k = search$ then
        $(\sigma_k, DB(w_k); EDB_k) \leftarrow$Search$(K, w_k, \sigma_{k-1}; EDB_{k-1})$
    else if $type_k = update$ then
        $(\sigma_k; EDB_k) \leftarrow$Update$(K, add/del, (id_k, w_k), \sigma_{k-1}; EDB_{k-1})$
    Let $t_k$ be the messages from client to server in the $Search/Update$ protocols above
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
return $b$;

$b \leftarrow \mathsf{Ideal}^{\mathsf{DSE}}_{\mathsf{Adv}, Sim, \mathcal{L}}(\lambda, q)$:
$N \leftarrow \mathsf{Adv}(1^\lambda)$
$(st_{\mathcal{S}}, EDB_0) \leftarrow$SimSetup$(1^\lambda, N)$
for $k = 1$ to $q$ do
    $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$
    if $type_k = search$ then
        $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow$SimSearch$(st_{\mathcal{S}}, \mathcal{L}^{Srch}(w_k); EDB_{k-1})$
    else if $type_k = update$ then
        $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow$SimUpdate$(st_{\mathcal{S}}, \mathcal{L}^{Updt}(w_k); EDB_{k-1})$
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
return $b$
```

Figure 8: Real and ideal experiments for the DSE scheme.

A symmetric-key encryption scheme is said to be CPA-secure if for all PPT adversaries Adv and any two arbitrary plaintext messages x_0 and x_1, $|Pr[\mathsf{Adv}(Enc_K(x_0)) = 1] - Pr[\mathsf{Adv}(Enc_K(x_1)) = 1]| \leq \nu(\lambda)$, where $K \leftarrow KeyGen(\lambda)$.

B DSE Security Game

Figure 8 shows the $\mathsf{Real}^{\mathsf{DSE}}$ and $\mathsf{Ideal}^{\mathsf{DSE}}$ games for the DSE security definition 1.

B.1 Leakage Functions for Backward Privacy

Assume Q is list with one entry for each operation. The entry for a search and an update is of the form (u, w) and $(u, op, (w, id))$, respectively, where u is the query timestamp (starting from 1), w is the searched keyword, $op = add/del$, and id is the modified file. For keyword w, $TimeDB(w) = \{(u, id) \mid (u, add, (w, id)) \in Q \wedge \forall u', (u', del, (w, id)) \notin Q\}$ returns the list of all timestamp/file-identifier pairs of keyword w that have been added to DB and have not been subsequently deleted. Finally, the function $DelHist(w) = \{(u^{add}, u^{del}) \mid \exists\, id : (u^{add}, add, (w, id)) \in Q \wedge (u^{del}, del, (w, id)) \in Q\}$ returns the history of deleted entries by giving all (insertion timestamp, deletion timestamp) pairs to the adversary. Informally, it shows which deletion corresponds to which addition.

C Formal Analysis

C.1 Proof of Lemma 1

Let s be the starting node of the search and recall that s is the root of a full binary tree. We denote by S the set of nodes in T_w that belong to the subtree rooted by s. Let $v_1, \ldots, v_l$ be the roots of the l full binary trees of the MCS forest for leafs $[1, i_w]$, with $l < \log i_w$. Let S' be the set that contains all the nodes of T_w that belong to a subtree rooted by any of the nodes v_i. Clearly, since s is a common ancestor of all v_i, $S' \subseteq S$. The set $S'' := S \setminus S'$ contains at most $\log i_w$ nodes that are ancestors of some of the leafs in $[1, i_w]$: one for each layer from s to right before the leafs (all parents of v_i).

Having defined these sets, the search process starts from s and accesses all the nodes in S'' and those in S' that have not been removed by some of the previous deletions. The number of nodes in S that are not deleted can be proven to be $\leq 2n_w$ by a simple induction on the number of deletions. For 0 deletions, S consists of full binary trees hence $|S| = 2i_w - 1 < 2n_w$, since $n_w = i_w$ in this case. After k deletions by the inductive hypothesis there will be less than $2(i_w - k)$ nodes. Since each deletion removes exactly two nodes from S (the leaf to be deleted and its parent), after deletion $k+1$, there will be less than $2(i_w - k) - 2 = 2(i_w - (k+1))$, i.e., less than twice the result size $(i_w - (k+1))$.

As S' and S'' are disjoint, the total number of nodes accessed by a search is upper bounded by $2n_w + \log i_w$. □

C.2 Proof of Security

Theorem 1. *Assuming F is a secure PRF, SKE is a secure encryption scheme, OM_{Del}, OM_{Tree} are secure oblivious maps,* OSSE *is adaptively-secure in the programmable random oracle model according to Definition 1, with $\mathcal{L}^{Setup}(N) = N$ and $\mathcal{L}^{Updt}(op, w, id) = op$ and $\mathcal{L}^{Srch}(w) = (TimeDB(w), DelHist(w))$.*

Proof. We prove the security of OSSE via a sequence of indistinguishable hybrids as follows:

- **Hybrid-0:** This is the Real^{SSE} game defined in Appendix B between the adversary and a challenger.

- **Hybrid-1:** This is the same as Hybrid-0 but all encryptions val corresponding to leaf nodes during insertion queries are replaced with $SKE.Enc(sk, 0)$, i.e., encryptions of zero. Since decryption happens only locally by the client, any adversary that can distinguish between these two games can be used to break the CPA security of *SKE* via a series of standard hybrids.

- **Hybrid-2:** This is the same as Hybrid-1 but all tokens *tk* for any keyword *w* and counter cnt_w are instead generated uniformly at random from the range of the PRF F $\{0,1\}^\lambda$. More specifically, the challenger initiates a map TK. Every time the code calls for a token *tk* for a new w, cnt_w combination, it gets the result of $TK.\mathrm{get}((w, cnt_w), tk)$. If the result is null, it chooses *tk* uniformly at random and stores it with $TK.\mathrm{put}((w, cnt_w), tk)$, for future reference. The two games can be shown to be indistinguishable due to the security of the PRF via a series of standard hybrids.

- **Hybrid-3:** This is the same as Hybrid-2 but the challenger internally maintains a list I that stores the contents of each query of the adversary (w for searches, (w, id, op) for updates). With this list I, the challenger can at any phase of the game, accurately compute the correct T_w tree for every keyword w, including the correct access counter for each node based on the history of the adversary's queries. This change is internal only so Hybrid-3 is identical to Hybrid-2 for the adversary.

- **Hybrid-4:** This is the same as Hybrid-3 but during setup the oblivious map initializations for OM_{Del}, OM_{Tree} are replaced with calls to two simulators SIM_{OMAP} for capacity N. All future oblivious map accesses are emulated by calls to the corresponding simulators SIM_{OMAP}. Whenever the code needs such an access during an update, the challenger recreates the corresponding tree T_w for the related keyword w at that phase of the game and calculates the access result from this. Hence, Hybrid-4 is indistinguishable from Hybrid-3 due to the security of the oblivious maps.

- **Hybrid-5:** This is the same as Hybrid-4 but all calls to H_0 are emulated by the challenger as follows. Whenever the code or the adversary calls for an evaluation of H_0, the challenger, who maintains a table H0 of all past H_0-calls, responds: (i) with a freshly chosen random value from the range of H_0 if this input has not been requested for evaluation before, in which case it stores it at H0 together with the input, (ii) with the evaluation result stored in H0 if the input has been requested before. Denote by Bad_1 the event where throughout the game the challenger happens to sample the same H_0 evaluation for different inputs. If Bad_1 takes place the challenger aborts. Clearly, conditioned on not aborting, the view provided to the adversary in Hybrid-4 is identical to that of Hybrid-3. Second, since the range of H_0 is assumed exponential to the security parameter λ, whereas the number of H_0-calls that will be executed throughout the game is polynomial in λ (since the adversary is bounded) by the birthday problem the probability of abort is negligible. Hence the games are statistically indistinguishable in the programmable random oracle model.

- **Hybrid-6:** This is the same as Hybrid-5 with the following modification. For calls to H_0 to calculate *addr* during insertions (Algorithm 1, line 4) and deletions (Algorithm 2, line 20), the challenger samples the evaluation as above but instead stores it to table H0$'$ together with the timestamp $1 \leq u \leq q$ of the operation. For deletions, it stores all $\log N$ sampled valued in the order in which they are called. Then, for a search query with tokens tk, tk_0, let $u_1, \ldots, u_t$ be the timestamps of all prior updates for the same keyword w, after the latest prior search for the same keyword. The challenger first builds all prior "snapshots" of $T_w^{(1)}, \ldots, T_w^{(t)}$. Based on this, it calculates for each update operation, the corresponding involved node labels, access counters, and the order in which they were accesses within the operation (for deletions). For each involved node label *lab* in these updates let *MaxAcc* be its maximal access counter in all tree snapshots. The challenger matches each such pair lab, acc for $acc \in [1, maxAcc]$ to the corresponding timestamp (and within the timestamped operation, the exact order to which this pair corresponds) and identifies the corresponding evaluation *addr* in H0$'$. It then removes this entry from H0$'$ and inserts to H0 the mapping $(tk, lab, acc) \rightarrow addr$. Note that, if $tk \neq tk_0$, entries for the latter have already been moved to H0. Effectively, this "matches" the evaluations during searched with the ones during previous related updates.
Denote by Bad_2 the event where when the challenger moves entries from H0$'$ to H0 in the above process, it encounters an already filled entry in H0. Since during the protocol execution all H_0 evaluations during searches follow prior evaluations on the same input during an update, this can only happen if the adversary has posted a query to H_0 for a token *tk* it has not yet seen, effectively guessing it. In this hybrid, the challenger modifies its code to check if Bad_2 take place; if that happens it aborts. First, note that unless the challenger aborts the game is identical to that of Hybrid-5. Second, given that tokens are sampled uniformly at random from a domain exponential in λ, whereas the total number of H_0-calls that the adversary can do throughout the game is polynomial in λ (since the adversary is PPT), the probability of aborting is negligible in λ, hence the two Hybrids are statistically indistinguishable.

- **Hybrid-7:** This is the same as Hybrid-6 but we now also replace H_1 with a programmable random oracle exactly as we did in Hybrid-5 for H_0. By the same reasoning as in that case, this hybrid is indistinguishable from the previous one.

- **Hybrid-8:** This is the same as Hybrid-7 but we now manipulate the evaluations of H_1 during deletions, exactly as we did in Hybrid-6 for H_1. More specifically, during deletions we sample the different *val* uniformly at random and store them at a table $\mathsf{H1}'$ for the time being. During searches, the only difference is that when moving an entry lab, acc with evaluation *val* and a corresponding token tk from $\mathsf{H1}'$ to $\mathsf{H1}$ during a search, tt sets the mapping to $(tk, lab, acc) \to val \oplus (left, leftAcc, right, rightAcc)$ where these four values can again be calculated directly from the series of trees $T_w^1, \ldots, T_w^t$. This ensures subsequent evaluations during the search execution by the server will be consistent with Algorithm 2, lines 20-21. Note that this also eliminates the need for token calculation during updates so this part is removed from the code of the challenger. By the same reasoning as for Hybrid-6, this hybrid is indistinguishable from the previous one.

- **Hybrid-9:** This is the same as Hybrid-8 but the challenger changes the list token map TK to map tokens to the search timestamp during they were created. This change is internal only so the game is identical to Hybrid-8 for the adversary.

- **Hybrid-10:** This is the same as Hybrid-9 but the challenger changes what the query list I store for each query. If k is an update, $I(k) = \perp$. If k is a search for keyword w, $I(k) = (TimeDB(w), DelHist(w))$.
  We note that the code of the challenger from Hybrid-9 can still be executed solely with this information, as follows. At this point, updates only involve sampling random appropriate random value for $addr, val$ and simulating oblivious map accesses. Regarding searches, first the challenger can find which prior operations involve the same keyword from $TimeDB(w), DelHist(w)$, directly for the case of prior updates, and indirectly by comparing $TimeDB(w)$ for past searches in order to reuse token tk_0. Second, it can calculate the counters cnt_w, i_w, d_w directly from $TimeDB(w), DelHist(w)$. Third, the functions $TimeDB(w), DelHist(w)$ map previous insertions and deletions to a unique T_w since they reveal when insertions took place and for each deletion specifically which prior insertion it cancels. Finally, from this the challenger can compute all nodes' access counters which are necessary for running the hybrid's search code. Again, this change is internal only so Hybrid-8 is identical to Hybrid-9 for the adversary.

- **Hybrid-11** This is the same as Hybrid-10 but the client only receives op (instead of op, w, id) during updates, and $TimeDB(w), DelHist(w)$ (instead of w) during searches. Clearly, this is again indistinguishable to the adversary.

The modified challenger's code for the final hybrid is essentially the code of our simulator in the Ideal^{SSE} game as it only takes as input the leakage of Theorem 1. Since we showed that Hybrid-0 and Hybrid 11 provide an indistinguishable view to an adversary playing the DSE adaptive security of Figure 8, the result follows.

□

We make two remarks about the above proof. First, even though $TimeDB(w)$ includes the actual id's of the documents in $DB(w)$ we never used this information, since our scheme's goal is to retrieve $DB(w)$ and this result remains always encrypted for the adversary. However, our use of $Time(DB)$ also covers our analysis for cases where OSSE is used in applications that want to retrieve the actual documents (see discussion in Section 3). Second, in the above we make the implicit assumption that search queries for non-existent keywords are not repeated; with this restriction, our leakage profile always captures standard search pattern leakage. However, to cover such cases, we should expand our leakage to explicitly include, when such a non-existing keyword was searched previously.

The following characterizes the security of LLSE (formal proof is deferred to the full version).

Theorem 2. *Assuming F is a secure PRF, SKE is a semantically secure encryption scheme, OM_{Del}, OM_{Tree} are secure oblivious maps,* LLSE *is adaptively-secure in the programmable random oracle model according to Definition 1, with $\mathcal{L}^{Setup}(N) = N$ and $\mathcal{L}^{Updt}(op, w, id) = op$ and $\mathcal{L}^{Srch}(w) = (TimeDB(w), DelHist(w))$.*