Practical Dynamic Searchable Encryption with Small Leakage

Emil Stefanov
UC Berkeley
emil@cs.berkeley.edu

Charalampos Papamanthou
University of Maryland
cpap@umd.edu

Elaine Shi
University of Maryland
elaine@cs.umd.edu

***Abstract*—Dynamic Searchable Symmetric Encryption (DSSE) enables a client to encrypt his document collection in a way that it is still searchable and efficiently updatable. However, all DSSE constructions that have been presented in the literature so far come with several problems: Either they leak a significant amount of information (e.g., hashes of the keywords contained in the updated document) or are inefficient in terms of space or search/update time (e.g., linear in the number of documents).**

In this paper we revisit the DSSE problem. We propose the first DSSE scheme that achieves the best of both worlds, i.e., both small leakage and efficiency. In particular, our DSSE scheme leaks *significantly less* information than any other previous DSSE construction and supports both updates and searches in sublinear time *in the worst case*, maintaining at the same time a data structure of only linear size. We finally provide an implementation of our construction, showing its practical efficiency.

I. Introduction

Searchable Symmetric Encryption (SSE) [31] enables a client to encrypt her document collection in a way that keyword search queries can be executed on the encrypted data via the use of appropriate "keyword tokens". With the advent of cloud computing (and the emerging need for privacy in the cloud), SSE schemes found numerous applications, e.g., searching one's encrypted files stored at Amazon S3 or Google Drive, without leaking much information to Amazon or Google. However, the majority of SSE constructions that have been presented in the literature work for static data: Namely there is a setup phase that produces an encrypted index for a specific collection of documents and after that phase, no additions or deletions of documents can be supported (at least in an efficient manner).

Due to various applications that the dynamic version of SSE could have, there has recently been some progress on Dynamic Searchable Symmetric Encryption (DSSE) [12], [20], [21], [36]. In a DSSE scheme, encrypted keyword searches should be supported even after documents are arbitrarily *inserted* into the collection or *deleted* from the collection. However, to assess the quality of a DSSE scheme, one must precisely specify the information *leakage* during searches and updates.

Minimizing the leakage for DSSE can be achieved by using ORAM [3], [10], [13], [15], [17]–[19], [23]–[25], [27], [30], [35], [37], [38] to hide every memory access during searches and updates. However, applying ORAM is costly in this setting (see Section II). In order to avoid expensive ORAM techniques, one could allow for some extra leakage. Ideally, the DSSE leakage should only contain:

a) The hashes of keywords we are searching for, referred to as *search pattern* in the literature [9].
b) The matching document identifiers of a keyword search and the document identifiers of the added/deleted documents, referred to as *access pattern* in the literature [9].
c) The current number of document-keyword pairs stored in our collection, which we call *size pattern*.

Note that the above DSSE leakage implies a strong property called *forward privacy*: If we search for a keyword w and later add a new document containing keyword w, the server does not learn that the new document has a keyword we searched for in the past. It also implies *backward privacy*, namely queries cannot be executed over deleted documents.

Unfortunately, existing *sublinear* DSSE schemes [20], [21], [36] not only fail to achieve forward and backward privacy, but also leak a lot of additional information during updates such as the keyword hashes shared between documents (not just the hashes of the queried keywords). Our main contribution is the construction of a new sublinear DSSE scheme whose leakage only contains (a), (b) and (c) from above (but, like any other existing scheme, it does not achieve backward privacy). In particular:

1) Our DSSE scheme has *small leakage*: Apart from the search, access and size patterns, it also leaks (during searches) the document identifiers that were deleted in the past and match the keyword. As such, our scheme achieves forward privacy (but not backward privacy).
2) Our DSSE scheme is *efficient*: Its worst-case search complexity is $O(\min\{\alpha + \log N, m \log^3 N\})$, where N

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/doi-info-to-be-provided-later

is the size of the document collection (specifically, the number of document-keyword pairs), m is the number of documents containing the keyword we are searching for and α is the number of times this keyword was *historically* added to the collection (i.e., for $\alpha = \Theta(m)$ the search complexity is $O(m + \log N)$; in any other case it cannot go beyond $O(m \log^3 N)$). The scheme's worst-case update complexity is $O(k \log^2 N)$, where k is the number of unique keywords contained in the document of the update (insertion or deletion). Finally, the space of our data structure is optimal (i.e., $O(N)$).

To the best of our knowledge, no other DSSE scheme in the literature achieves Properties (1) and (2) simultaneously. See the paragraph of Section II that refers to DSSE related work for details. Further contributions of our work include:

3) Our DSSE scheme is the first one to support dynamic keywords. As opposed to previous DSSE schemes that require storing information about all the possible keywords that *may* appear in the documents (i.e., all the dictionary), our scheme stores only information about the keywords that *currently* appear in the documents.
4) We implement our DSSE scheme on memory and we show that our scheme is very efficient in practice, achieving a query throughput of 100,000 search queries per second (for result size equal to 100). To achieve practical efficiency, our implementation is fully parallel and asynchronous and we can parallelize queries along with updates at the same time.

Technical highlights. Our technique departs from existing index-based techniques for SSE (e.g., [9], [20], [21], [31], [36]) that use an encrypted inverted index data structure. Instead it stores document-keyword pairs in a hierarchical structure of logarithmic levels, which is reminiscent of algorithmic techniques used in the ORAM literature (e.g., [15], [16], [32], [34], [37]). A similar structure was also recently used by the authors to construct very efficient dynamic Proofs of Retrievability [33].

Specifically, in our scheme, when a document x containing keyword w is added to our collection, we store in a hash table an encryption of the tuple (w, x, add, i), where i is a counter indicating that x is the i-th document containing keyword w. When a document x containing keyword w is deleted from our collection, an encryption of the tuple (w, x, del, i) is stored. During the encrypted search for a certain keyword w, all hash table keys of addition/deletion entries referring to w are retrieved (and decrypted) via an appropriate token for w (generated by the client).

Storing both addition and deletion entries can however lead to linear worst-case complexity for search, e.g., first we add some documents containing keyword w, then we delete all documents containing w and then we search for keyword w. In this case the search will have to iterate through all the addition/deletion entries in order to conclude that no document contains keyword w.

To avoid this scenario, we need to rebuild the data structure periodically (so that opposite entries can be canceled out), which is again a linear cost. To reduce that rebuilding cost from linear to logarithmic, we use the multilevel structure that we mentioned above (instead of storing everything in a flat hash table). Forward privacy is derived from the fact that every time we rebuild a level of the above data structure, we use a *fresh* key for encrypting the entries within the new level—this makes old tokens unusable within the new level.

II. Related Work

Static SSE. In the static setting, Curtmola et al. [9] gave the first index-based SSE constructions to achieve sublinear search time. A similar construction was also described by Chase and Kamara [8], but with higher space complexity. Finally, recent work by Kurosawa et al. [22] shows how to construct a (verifiable) SSE scheme that is universally composable (UC). While UC-security is a stronger notion of security, their construction requires linear search time. Finally, Cash et al. [5] recently presented an SSE scheme for conjunctive queries over static data. An extension of this protocol that allows the data owner to authorize third parties to search in the encrypted static database was recently also proposed by Cash et al. [6].

Dynamic SSE. Song et al. [31] were the first to explicitly consider the problem of searchable encryption and presented a scheme with search time that is linear in the size of the data collection. Their construction supports insertions/deletions of files in a straightforward way. Goh [12] proposed a dynamic solution for SSE, which again requires linear search time and results in false positives. Chang and Mitzenmacher [7] proposed a construction with linear search time but without false positives—their solution also achieves forward privacy.

The recently introduced dynamic scheme of Kamara et al. [21] was the first one with sublinear search time, but it does not achieve forward privacy and reveals hashes of the unique keywords contained in the document of the update. The scheme of Kamara and Papamanthou [20] overcomes the above limitation (still not achieving forward privacy) by increasing the space of the used data structure.

Finally, the work of van Liesdonk et al. [36] has the limitations of both [21] and [20], with leaky updates and a large index. Also, the number of updates supported by their scheme is not arbitrary.

Dynamic SSE through ORAM. The DSSE problem can be solved by using oblivious RAM (ORAM) [3], [10], [13], [15], [17]–[19], [23]–[25], [27], [30], [35], [37], [38] as a black box. ORAM provides the strongest levels of security, namely the server only learns the size of the document collection.

However, ORAM schemes are less efficient in practice due to a big overhead in terms of bandwidth. The ORAM schemes that achieve low bandwidth (e.g., [32], [34], [35] rely on the block sizes being relatively large (e.g., 4 KB).

In order to handle small block sizes such as document-keyword pairs, those ORAM techniques would end up using a lot more client storage (because the ORAM would consist of many more but smaller blocks). Those schemes can be re-parameterized to use less client storage (e.g., by using recursion as in [34], [35]), but that would drastically increase bandwidth and might in some cases result in multiple round-trips of communication.

Other related work. Related to searchable encryption is also functional encryption [4], [11], [28], where one encrypts the documents in a way that one can issue tokens that would allow testing whether a specific keyword is contained in the document, without decrypting the document. However such solutions incur linear cost for search (however it is straightforward to address updates). Aiming at more efficient schemes, Boneh et al. [2] presented functional encryption schemes for specific functionalities such as keyword search and Shi et al. [29] presented functional encryption schemes for multidimensional queries, with linear cost for searches.

III. Preliminaries

The notation $((c\_out, s\_out) \leftarrow \mathsf{protocol}((c\_in, s\_in))$ is used to denote a protocol between a client and a server, where *c_in* and *c_out* are the client's input and output; *s_in* and *s_out* are the server's input and output.

Definition 1 (DSSE scheme)**.** *A dynamic searchable symmetric encryption (DSSE) scheme is a suite of three protocols that are executed between a client and a server with the following specification:*

- $(\mathsf{st}, D) \leftarrow \mathsf{Setup}((1^\lambda, N), (1^\lambda, \perp))$. *On input the security parameter λ and the number of document-keyword pairs N, it outputs a secret state* st *(to be stored by the client) and a data structure D (to be stored by the server);*
- $((\mathsf{st}', \mathcal{I}), \perp) \leftarrow \mathsf{Search}((\mathsf{st}, w), D)$. *The client input's include its secret state* st *and a keyword w; and server's input is its data structure D. At the end of the* Search *protocol, the client outputs a possibly updated state* st' *and the set of document identifiers $\mathcal{I}$ that contain the keyword w. The server outputs nothing.*
- $(\mathsf{st}', D') \leftarrow \mathsf{Update}((\mathsf{st}, \mathsf{upd}), D)$. *The client has input* st*, and an update operation* $\mathsf{upd} := (\mathsf{add}, \mathsf{id}, \mathbf{w})$ *or* $\mathsf{upd} := (\mathsf{del}, \mathsf{id}, \mathbf{w})$ *where* id *is the document identifier to be added or removed, and* $\mathbf{w} := (w_1, w_2, \ldots, w_k)$ *is the list of unique keywords in the document. The server's input is its data structure D. The* Update *protocol adds (or deletes) the document to (or from) D, and results in an updated client secret state* st' *and an updated server data structure D'.*

We note that in our construction, Setup and Search can be performed *non-interactively, i.e., involving only a single-round trip* between the server and client. In our construction the Update protocol is *interactive*, but the client can always answer a search query in a single round by storing a small buffer of documents currently being updated in the background until the updates finish.

A. Security Definition

We define security using the standard simulation model of secure computation [14], requiring that a real-world execution "simulates" an ideal-world (reactive) functionality. For clarity of presentation, we first present a scheme secure in the semi-honest model, where the adversary (i.e., server) faithfully follows the prescribed protocol, but is curious. Then, in Section VI we show how to make our protocol work in the malicious model as well. We now define the following experiments:

Ideal-world execution $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$**.** An environment $\mathcal{Z}$ sends the client a message "*setup*". The client then sends an ideal functionality $\mathcal{F}$ a message "*setup*". The ideal-world adversary $\mathcal{S}$ (also referred to as a simulator) is notified of N, an upper bound on the number of document-keyword pairs.

In each time step, the environment $\mathcal{Z}$ specifies a search or update operation to the client. For a search operation, $\mathcal{Z}$ picks a keyword w to search. For an update operation, $\mathcal{Z}$ picks $\mathsf{upd} := (\mathsf{add}, \mathsf{id}, \mathbf{w})$ or $\mathsf{upd} := (\mathsf{del}, \mathsf{id}, \mathbf{w})$. The client sends the search or update operation to the ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies $\mathcal{S}$ of $\mathsf{leak}_s(w)$ for a search operation, and $\mathsf{leak}_u(\mathsf{upd})$ for an update operation (see Section III-B for the definition of the leakage functions). $\mathcal{S}$ sends $\mathcal{F}$ either abort or continue. As a result, the ideal-functionality $\mathcal{F}$ sends the client $\perp$ (to indicate abort), "update success", or the indices of matching documents for a search query. The environment $\mathcal{Z}$ gets to observe these outputs.

Finally, the environment $\mathcal{Z}$ outputs a bit $b \in \{0, 1\}$.

Real-world execution $\textsc{Real}_{\Pi_\mathcal{F},\mathcal{A},\mathcal{Z}}$**.** An environment $\mathcal{Z}$ sends the client a message "*setup*". The client then performs the Setup protocol (with input N) with the real-world adversary $\mathcal{A}$.

In each time step, an environment $\mathcal{Z}$ specifies a search or update operation to the client. For a search operation, $\mathcal{Z}$ picks a keyword w to search. For an update operation, $\mathcal{Z}$ picks $\mathsf{upd} := (\mathsf{add}, \mathsf{id}, \mathbf{w})$ or $\mathsf{upd} := (\mathsf{del}, \mathsf{id}, \mathbf{w})$. The client then executes the real-world protocols Search or Update with the server on the inputs chosen by the environment. The environment $\mathcal{Z}$ can observe the client's output in each time step, which is either $\perp$ (indicating protocol abortion), "update success", or the indices of matching documents to a search query.

Finally, the environment $\mathcal{Z}$ outputs a bit $b \in \{0, 1\}$.

Definition 2 (Semi-honest/malicious security)**.** *We say that a protocol $\Pi_\mathcal{F}$ emulates the ideal functionality $\mathcal{F}$ in the semi-honest (or malicious) model, if for any probabilistic, polynomial-time semi-honest (or malicious) real-world adversary $\mathcal{A}$, there exists an simulator $\mathcal{S}$, such that for all non-uniform, polynomial-time environments $\mathcal{Z}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$|\Pr[\textsc{Real}_{\Pi_\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)\,.$$

Specifically, in the above definition, a semi-honest adversary always faithfully follows the prescribed protocol, whereas a malicious adversary can arbitrarily deviate from the protocol. Informally, with a malicious adversary, we would like to detect any deviation from the prescribed protocol, i.e., any deviation can be detected and is equivalent to aborting.

The above definition simultaneously captures correctness and privacy. Correctness is captured by the fact that the ideal-world client either receives the correct answer of the query, or receives an abort message. Privacy is captured by the fact that the ideal-world adversary (i.e., simulator) has no knowledge of the client's queries or dataset, other than the leakage explicitly being given to the simulator.

B. Defining Leakage

An update $\mathsf{upd} = (\mathsf{op}, \mathsf{id}, \mathbf{w})$ leaks the type of the update op, the identifier of the document id that is being updated, the number of keywords $|\mathbf{w}|$ in the document and the time t of of the update (i.e., when a document is added or removed). Therefore we define $\mathsf{leak}_u(\mathsf{upd}) = [\mathsf{op}, \mathsf{id}, |\mathbf{w}|, t]$. As opposed to [21], our update protocols do not leak *which* keywords are contained in the updated file.

A search for a keyword w_i leaks a set $\mathcal{I}$ containing the identifiers of documents matching keyword w_i that were added or removed in the past (referred to as access pattern in [21]). It also leaks a vector v_{w_i} of i entries such that $v_{w_i}(j) = 1$ iff there was a search for w_i at time $j < i$ (referred to as search pattern in [21]). Therefore we define $\mathsf{leak}_s(w_i) = [\mathcal{I}, v_{w_i}]$.

Note that this definition of leakage captures *forward privacy*, in that the set of indices matching $\mathcal{I}$ leaked contains only documents that were added in the past, but no future documents. Our definition of leakage does not satisfy *backward privacy*, since the set of matching $\mathcal{I}$ leaked also contains documents that were previously added but then deleted.

IV. Basic Construction

In this section, we first describe a (relatively inefficient) basic scheme whose search complexity is linear in the number of documents that have been historically added, containing the keyword w. Later, in Section V, we describe a new technique to reduce the search cost to roughly the number of matching documents, instead of all documents that have been historically added containing the keyword—note that some of these documents may already have been removed at the time of the search. Our second construction is built on top of the basic one and we present it as two parts for clarity.

A. Server-Side Data Structure

Hierarchical levels. The server stores a hierarchical data structure containing $\log N + 1$ *levels*, denoted $\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_L$, where $L = \log N$. For each level $0 \le \ell \le L$, level ℓ can store up to 2^ℓ *entries*. Each entry encodes the information $(w, \mathsf{id}, \mathsf{op}, cnt)$, where w is a keyword; op encodes the op-code taking a value of either add or del; id is a document identifier containing the keyword w; and cnt denotes the current counter for keyword w within level $\mathbf{T}_\ell$.

Intuitively, we can think of each level $\mathbf{T}_\ell$ as the permuted encoding of a table Γ_ℓ. Henceforth, we use the notation Γ_ℓ to denote the *conceptual, unencoded data structure* at level ℓ, and we use the notation $\mathbf{T}_\ell$ to denote the *encoded table* at level ℓ that is actually stored on the server.

The conceptual data structure at level ℓ. The conceptual, unencoded data structure at level ℓ is as follows:

$$\Gamma_\ell : w \to \left[\begin{array}{l} (\mathsf{id}, \mathsf{add}, 0), (\mathsf{id}, \mathsf{add}, 1), \ldots, (\mathsf{id}, \mathsf{add}, cnt_{\mathsf{add}}), \\ (\mathsf{id}, \mathsf{del}, 0), (\mathsf{id}, \mathsf{del}, 1), \ldots, (\mathsf{id}, \mathsf{del}, cnt_{\mathsf{del}}) \end{array} \right]$$

In other words, for each word w, each level stores add and del operations associated with the word w: Specifically, an $(\mathsf{id}, \mathsf{add}, cnt)$ tuple means that a document identified by id containing the word w is added; an $(\mathsf{id}, \mathsf{del}, cnt)$ tuple means that a document identified by id containing the word w is deleted.

We ensure that within the same level the same (w, id) pair only appears once for an add operation or a del operation, but not both—if both appear, they cancel each other out during the level rebuilding as explained later.

Furthermore, we ensure that all $(w, \mathsf{id}, \mathsf{op}, cnt)$ tuples are lexicographically sorted based on the key $(w, \mathsf{id}, \mathsf{op})$.

Encoding the conceptual data structure. The conceptual table Γ_ℓ is encoded and its entries are then permuted, before being stored on the server. We now describe how to encode each table Γ_ℓ satisfying the following requirements.

- *Confidentiality.* The idea is to "encrypt" it in such a way such that it does not leak any information normally.
- *Tokens allow conditional decryption.* However, when the client needs to search a keyword w, it can release a token $\mathsf{token}_\ell(w)$ for each level ℓ, such that the server can then decrypt all entries $\Gamma_\ell[w]$ without learning any additional information. Each token corresponds to a (keyword, level) pair.
- *Constant table lookup time.* Not only can the server decrypt all entries in $\Gamma_\ell[w]$, given $\mathsf{token}_\ell(w)$, the server can read $\mathsf{id} := \Gamma_\ell[w, \mathsf{op}, cnt]$ in $O(1)$ time.

B. Algorithms for Encoding the Level Data Structure

We now explain how to encode a conceptual table Γ_ℓ into an encoded table $\mathbf{T}_\ell$ that is actually stored on the server. For this the client is going to use a secret key k_i for each level $i = 0, 1, \ldots, \ell$ and a secret key esk to be used for a randomized symmetric encryption scheme.

The conceptual table Γ_ℓ can be thought of as a collection of entries each encoding a tuple $(w, \mathsf{id}, \mathsf{op}, cnt)$. Each entry will be encoded using the $\mathsf{EncodeEntry}$ algorithm as described in Figure 1. When an encoded level $\mathbf{T}_\ell$ is being

Algorithm $\mathsf{EncodeEntry}_{esk,k_\ell}(w, \mathsf{id}, \mathsf{op}, cnt)$

1. $\mathsf{token}_\ell := \mathsf{PRF}_{k_\ell}(h(w))$.
2. $\mathsf{hkey} := \mathsf{H}_{\mathsf{token}_\ell}(0||\mathsf{op}||cnt)$.
3. $c_1 := \mathsf{id} \oplus \mathsf{H}_{\mathsf{token}_\ell}(1||\mathsf{op}||cnt)$.
4. $c_2 := \mathsf{Encrypt}_{esk}(w, \mathsf{id}, \mathsf{op}, cnt)$.
5. Output $(\mathsf{hkey}, c_1, c_2)$.

Fig. 1: The algorithm for encoding an entry.

Algorithm $\mathsf{Lookup}(\mathsf{token}, \mathsf{op}, cnt)$

1. $\mathsf{hkey} := \mathsf{H}_{\mathsf{token}}(0||\mathsf{op}||cnt)$.
2. If $\mathsf{hkey} \notin \mathbf{T}_\ell$, output $\perp$.
3. Else, output $\mathsf{id} := \mathbf{T}_\ell[\mathsf{hkey}].c_1 \oplus \mathsf{H}_{\mathsf{token}}(1||\mathsf{op}||cnt)$.

Fig. 2: The algorithm for looking up an entry.

built, all these encoded entries are randomly permuted by the client (using an oblivious sorting algorithm as explained later). Our construction is using a keyed hash function $\mathsf{H}_k : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ (this is modeled as a random oracle in our proof of security in Section VIII-B—we show how to avoid using the random oracle in Section VI by increasing the client computation and the communication). Also, we use $h(\cdot)$ to denote a standard hash function (e.g., SHA256).

All encoded and permuted entries are stored on the server in a hash table indexable by hkey, denoted

$$\mathbf{T}_\ell[\mathsf{hkey}] = (c_1, c_2)\,.$$

This hash table allows the server to achieve constant lookup: With an appropriate token for a keyword w, the server can decrypt the c_1 part, and read off the entry $\mathsf{id} := \Gamma[w, \mathsf{op}, cnt]$ in $O(1)$ time. The Lookup algorithm is detailed in Figure 2.

We also quickly note that the term c_2 in the above will later be used by the client during updates.

C. Basic Searchable Encryption Scheme

We now give the detailed description (Figure 3) of the three protocols that constitute a dynamic searchable encryption scheme, as defined in Definition 1.

D. Rebuilding Levels During Updates

Every update entry that the client sends to the server causes a rebuild of levels in the data structure. The basic idea of the rebuild is to take consecutively full levels $\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_{\ell-1}$, as well as a newly added entry, and merge them (e.g., via protocol SimpleRebuild) into the first empty level $\mathbf{T}_\ell$. Since protocol SimpleRebuild has $O(N \log N)$ complexity (see below), it is easy to see that every update takes $O(\log^2 N)$ amortized time, for large levels are rebuilt

Protocol $(\mathsf{st}, D) \leftarrow \mathsf{Setup}((1^\lambda, N), (1^\lambda, \perp))$

Client chooses an encryption key esk, and $L = \log N$ random level keys $k_0, k_1, \ldots, k_L$. The secret client state consists of $\mathsf{st} := (esk, k_0, k_1, \ldots, k_L)$.
Server allocates an empty hierarchical structure D, consisting of exponentially growing levels $\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_L$.

Protocol $((\mathsf{st}', \mathcal{I}), \perp) \leftarrow \mathsf{Search}((\mathsf{st}, w), D)$

1) *Client:* Given a keyword w, the client computes a token for each level

$$\mathsf{tks} := \{\mathsf{token}_\ell := \mathsf{PRF}_{k_\ell}(h(w)) : \ell = 0, 1, \ldots, L\}.$$

The client sends the tokens tks to the server.
2) *Server:* Let $\mathcal{I} := \emptyset$. For $\ell \in \{L, L-1, \ldots, 0\}$ do:
   - For $cnt := 0, 1, 2, 3, \ldots$ until not found:
     $\mathsf{id} := \mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{add}, cnt)$.
     $\mathcal{I} := \mathcal{I} \cup \{\mathsf{id}\}$.
   - For $cnt := 0, 1, 2, 3, \ldots$ until not found:
     $\mathsf{id} := \mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{del}, cnt)$.
     $\mathcal{I} := \mathcal{I} - \{\mathsf{id}\}$.

   Return $\mathcal{I}$ to the client.

Protocol $(\mathsf{st}', D') \leftarrow \mathsf{Update}((\mathsf{st}, \mathsf{upd}), D)$

Let $\mathsf{upd} := (\mathbf{w}, \mathsf{id}, \mathsf{op})$ denote an update operation, where $\mathsf{op} = \mathsf{add}$ or $\mathsf{op} = \mathsf{del}$ and $\mathbf{w}$ is the vector storing the unique keywords contained in the document of identifier id.
For $w \in \mathbf{w}$ in random order do:

- If $\mathbf{T}_0$ is empty, select a fresh key k_0 and set $\mathbf{T}_0 := \mathsf{EncodeEntry}_{esk,k_0}(w, \mathsf{id}, \mathsf{op}, cnt = 0)$.
- Else, let $\mathbf{T}_\ell$ denote the first empty level:
  Call $\mathsf{SimpleRebuild}(\ell, (w, \mathsf{id}, \mathsf{op}))$.
  (or $\mathsf{Rebuild}(\ell, (w, \mathsf{id}, \mathsf{op}))$).

Fig. 3: Our basic construction.

a lot less frequently than small levels (note however that the bandwidth required for each update is $O(\log N)$). In particular, over a course of $N = 2^\ell$ operations, level 0 is rebuilt $N/2$ times, level 1 is rebuilt $N/4$ times and level $\ell - 1$ is rebuilt $N/2^\ell = 1$ times.

We note here that we can use standard de-amortization techniques (e.g., [15], [16], [37]) to turn these complexities into worst-case. Namely, each update in our scheme (and in our implementation) induces $O(\log N)$ bandwidth and takes $O(\log^2 N)$ time in the *worst-case*.

Concerning the SimpleRebuild protocol in Figure 4, note the sorting performed in Step 3 before the entries are processed in Step 4: The reason for this sorting is to ensure that Step 4 can be performed in $O(N)$ time via a sequential scan (instead of $O(N^2)$ time). Finally, there is another sorting taking place in Step 5 (based on hkey) before uploading the entries to the server, and after they have been processed in Step 4. This performs a random shuffle, ensuring that the order of the encrypted entries do not reveal any information about the order of the underlying plaintexts.

Protocol $\mathsf{SimpleRebuild}(\ell, (w, \mathsf{id}, \mathsf{op}))$

(Assuming $O(N)$ client working storage)

1) Client creates local buffer $\mathbf{B} = (w, \mathsf{id}, \mathsf{op}, cnt = 0)$.
2) For each $\mathsf{entry} = (\mathsf{hkey}, c_1, c_2) \in \mathbf{T}_0 \cup \mathbf{T}_1 \cup \ldots \cup \mathbf{T}_{\ell-1}$:
   Let $(w, \mathsf{id}, \mathsf{op}, cnt) := \mathsf{Decrypt}_{esk}(c_2)$.
   Let $\mathbf{B} := \mathbf{B} \cup (w, \mathsf{id}, \mathsf{op}, cnt)$.
   // Client: download and decrypt all entries, store in local $\mathbf{B}$.
3) Sort $\mathbf{B}$ based on lexicographical sorting key $(w, \mathsf{id}, \mathsf{op})$.
   // All entries with the same keyword now appear sequentially.
4) For each $\mathbf{e} := (w, \mathsf{id}, \mathsf{op}, cnt') \in \mathbf{B}$ (in sorted order):
   - If $\mathbf{e}$ marks the start of a new word w, for an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, then set $cnt_{\mathsf{op},w} := 0$ and update $\mathbf{e} := (w, \mathsf{id}, \mathsf{op}, 0)$ in $\mathbf{B}$.
   - If $\mathbf{e}$ and its adjacent entry are add and del operations for the same (w, id) pair, suppress the entries by updating both entries with $\perp$.
   - Else, update $\mathbf{e} := (w, \mathsf{id}, \mathsf{op}, cnt_{\mathsf{op},w}\mathsf{++})$ in $\mathbf{B}$.
5) Select a fresh new level key k_ℓ.
   $\mathbf{T}_\ell := \{\mathsf{EncodeEntry}_{esk,k_\ell}(\mathsf{entry})\}_{\mathsf{entry} \in \mathbf{B}}$.
   // Dummy entries marked $\perp$ *are also encoded as part of* $\mathbf{T}_\ell$.
   Upload $\mathbf{T}_\ell$ to the server in the order of increasing hkey. Empty all the old levels $\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_{\ell-1}$.

Fig. 4: The simple rebuilding algorithm.

Protocol $\mathsf{Rebuild}(\ell, (w, \mathsf{id}, \mathsf{op}))$

(Assuming $O(N^\alpha)$ client working storage, $0 < \alpha < 1$)

1) Let $\mathsf{entry}^* := \mathsf{EncodeEntry}_{esk,k_0}(w, \mathsf{id}, \mathsf{op}, cnt = 0)$.
   Let $\hat{\mathbf{B}} := \{\mathsf{entry}^*\} \cup \mathbf{T}_0 \cup \mathbf{T}_1 \cup \ldots \cup \mathbf{T}_{\ell-1}$.
2) For each $\mathsf{entry} = (\mathsf{hkey}, c_1, c_2) \in \hat{\mathbf{B}}$:
   Let $(w, \mathsf{id}, \mathsf{op}, cnt) := \mathsf{Decrypt}_{esk}(c_2)$.
   Overwrite entry with $\mathsf{Encrypt}_{esk}(w, \mathsf{id}, \mathsf{op}, cnt)$.
   // Wrap entries in $\hat{\mathbf{B}}$ *in a randomized encryption scheme to prepare for oblivious sorting. During the execution of oblivious sorting, an entry is re-encrypted in a randomized fashion each time upon write.*
3) $\hat{\mathbf{B}} := \textbf{o-sort}(\hat{\mathbf{B}})$, based on the lexicographical sorting key $(w, \mathsf{id}, \mathsf{op})$.
   // Now, all entries with the same keyword appear sequentially.
4) For each entry $\mathbf{e} := \mathsf{Encrypt}_{esk}(w, \mathsf{op}, \mathsf{id}, cnt') \in \hat{\mathbf{B}}$ (in sorted order):
   - If $\mathbf{e}$ marks the start of a new word w, for an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, then set $cnt_{\mathsf{op},w} := 0$ and update $\mathbf{e} := \mathsf{Encrypt}_{esk}(w, \mathsf{id}, \mathsf{op}, 0)$ in $\hat{\mathbf{B}}$.
   - If $\mathbf{e}$ and its adjacent entry are add and del operations for the same (w, id) pair, suppress the entries by updating both entries with $\mathsf{Encrypt}_{esk}(\perp)$.
   - Else, update $\mathbf{e} := \mathsf{Encrypt}_{esk}(w, \mathsf{id}, \mathsf{op}, cnt_{\mathsf{op},w}\mathsf{++})$ in $\hat{\mathbf{B}}$.
5) Randomly permute $\hat{\mathbf{B}} := \textbf{o-sort}(\hat{\mathbf{B}})$, based on hkey.
6) Select a new level key k_ℓ.
   For each $\mathsf{entry} \in \hat{\mathbf{B}}$:
   $(w, \mathsf{id}, \mathsf{op}, cnt) := \mathsf{Decrypt}_{esk}(\mathsf{entry})$.
   Add $\mathsf{EncodeEntry}_{esk,k_\ell}(w, \mathsf{id}, \mathsf{op}, cnt)$ to $\mathbf{T}_\ell$.

Fig. 5: The main rebuilding algorithm.

For clarity, we first describe in Figure 4 the SimpleRebuild algorithm for the case when the client has sufficient (i.e., linear in the size of the dataset) local storage. Then, in Figure 5 we will describe protocol Rebuild for the same purpose. The difference between the SimpleRebuild and the Rebuild protocol is that in the SimpleRebuild protocol, the client downloads the entire level from the server, locally computes the result, and then uploads the result to the server. In the Rebuild protocol, we assume that the client has small local storage, and is not able to download the entire level at once.

In this case (i.e., when the client has small local storage), the most straightforward approach would be to treat the server as a remote Oblivious RAM (ORAM). The

client computes everything in the same manner as before—however, instead of accessing its local memory, the client now reads and writes data from and to the server's ORAM as necessary, in an oblivious manner. However, as we argued in Section II, ORAM is not practical for the case of small blocks used by our scheme.

We note that the Rebuild protocol of Figure 5 uses an oblivious sorting protocol to optimize the above generic ORAM scheme.

E. Oblivious Sorting

The oblivious sorting (**o-sort**) algorithm used in Figure 5 reorders the entries in the level. The **o-sort** algorithm allows the client to re-sort the entries based on their plaintext values without the server learning the plaintexts and their order before and after sorting.

We use a standard **o-sort** algorithm (e.g., see the algorithm described in [17]). On a high level, **o-sort** works as follows. The client downloads a small subsets of the entries, sorts them locally, and uploads them back to the server (possibly to different locations). After several rounds of sorting subsets of entries, the entire level becomes sorted.

Our construction ensures that during sorting entries are wrapped in a non-deterministic encryption layer. Every time an entry is uploaded back to the server, it is re-encrypted with a different nonce so the server cannot link the positions of the reordered entries to their original and intermediate positions before and during sorting.

Furthermore, locations of the entries that the client accesses are independent of the entry values and their sorting order, so the server is not able to learn anything about the entries by observing the **o-sort**.

We use an **o-sort** algorithm that uses $O(N^a)$ client memory for $0 < a < 1$ (e.g., about 210 MB of client memory in our experiments—see Section VII) to obliviously sort N elements with $O(N)$ entry I/O operations and $O(N \log N)$ client-side computation. Note that even though the number of entry I/O operations is $O(N)$, the number of comparison performed by the client is still $O(N \log N)$ so this does not violate the lower bound for sorting. For a more detailed description of the **o-sort** algorithm please refer to [17].

V. Sublinear Construction

As mentioned in the introduction, we would like to achieve at most $O(m \log^3 N)$ cost for search queries, where m is the number of matching documents. Unfortunately, our basic construction described in Section IV fails to achieve this. Consider the following counter-example:

Example. Suppose documents $1, 2, 3, \ldots \kappa$ are added ($\kappa = O(N)$), all of which contain a keyword w. Then, suppose documents $1, 2, 3, \ldots, \kappa - 1$ are deleted. At this point, running a search query on w should return only one document (document κ). However, in our basic construction, it is possible that the top level contains the add operations for documents $1, 2, 3, \ldots \kappa$, and then the lower levels contain the del operations for documents $1, 2, 3, \ldots \kappa - 1$. In this case, the search query would take time $\tilde{O}(\kappa)$ (which can be as large as $\tilde{O}(N)$), even though the result set is of size 1.

In this section, we will extend the basic construction to guarantee that a search operation takes sublinear time $O(m \log^3 N)$ even in the worst case, where m is the number of matching documents.

A. Extensions to the Main Construction

In the main protocol, we store the tuple $(w, \mathsf{id}, \mathsf{op}, cnt)$, for each (w, id) pair that was either added or deleted. In the following we describe one extra piece of information that we need to store with each tuple.

Storing target levels. In the new protocol, with each tuple $(w, \mathsf{id}, \mathsf{op}, cnt)$ stored at the level ℓ of the data structure, we are going to store the *target level* ℓ^* such that

- If $\mathsf{op} = \mathsf{add}$, then ℓ^* is the level of the data structure that the tuple $(w, \mathsf{id}, \mathsf{op}, cnt)$ is stored, i.e., $\ell^* = \ell$;
- If $\mathsf{op} = \mathsf{del}$, then ℓ^* is the level of the data structure that the respective *addition tuple* $(w, \mathsf{id}, \mathsf{add}, cnt)$ is stored, i.e., $\ell^* > \ell$ (since deletions happen only after additions).

We note here that the client can easily compute the target level of each new entry $(w, \mathsf{id}, \mathsf{op})$. Specifically, if $\mathsf{op} = \mathsf{add}$, the target level ℓ^* (which is the level of the entry) is the first empty level. Otherwise (i.e., when $\mathsf{op} = \mathsf{del}$), the client can retrieve the timestamp of the respective addition entry $(w, \mathsf{id}, \mathsf{add})$ and can compute its level ℓ (which is the target level of $(w, \mathsf{id}, \mathsf{del})$), since the data structure is built deterministically and is not dependent on the values of w and id.

Finally, we are going to maintain the invariant that all tuples $(\ell^*, w, \mathsf{id}, \mathsf{op}, cnt)$ with target level ℓ^* stored at level ℓ are going to be lexicographically sorted based on the key $(\ell^*, w, \mathsf{id}, \mathsf{op})$ (instead of just $(w, \mathsf{id}, \mathsf{op})$).

New encoding. Note now that the target level for each entry is encoded in the same way as the identifier id. Basically, for an entry $(w, \mathsf{id}, \mathsf{op})$ of level ℓ and target level ℓ^*, we modify Line 3 of the EncodeEntry algorithm to the following:

$$c_1 := (\ell^*, \mathsf{id}) \oplus \mathsf{PRF}_{\mathsf{token}_\ell}(1 || \mathsf{op} || cnt)\,,$$

where ℓ is the level of the tuple $(\ell^*, w, \mathsf{id}, \mathsf{op})$. In this way, given an appropriate token for a keyword w and level ℓ, the server can decrypt the entry

$$(\ell^*, \mathsf{id}) := \Gamma_\ell[w, \mathsf{op}, cnt]$$

in constant time. For simplicity, we write the new tuple as

$$(\ell^*, w, \mathsf{id}, \mathsf{op}, cnt)\,.$$

An illustrative example. Consider the following state of the data structure with the old encoding $(w, \mathsf{id}, \mathsf{op}, cnt)$, where

the tuples appear in lexicographic ordering based on the key $(w, \mathsf{id}, \mathsf{op})$ (note that we show *some* of the entries of the levels and not *all* the entries of the levels):

- Level 5:
  $(w, 1, \mathsf{add}, 0), (w, 4, \mathsf{add}, 1), (w, 13, \mathsf{add}, 2)$
- Level 4:
  $(w, 3, \mathsf{add}, 0)$
- Level 3:
  $(w, 1, \mathsf{del}, 0), (w, 3, \mathsf{del}, 1), (w, 4, \mathsf{del}, 2), (w, 19, \mathsf{add}, 0)$

In the above example, documents $1, 4, 13, 3$ containing keyword w were added, then documents $1, 3, 4$ were deleted and finally document 19 containing keyword w was added. With the new encoding $(\ell^*, w, \mathsf{id}, \mathsf{op}, cnt)$, the levels are going to be as below (the tuples appear in lexicographic ordering based on the key $(\ell^*, w, \mathsf{id}, \mathsf{op})$):

- Level 5:
  $(5, w, 1, \mathsf{add}, 0), (5, w, 4, \mathsf{add}, 1), (5, w, 13, \mathsf{add}, 2)$
- Level 4:
  $(4, w, 3, \mathsf{add}, 0)$
- Level 3:
  $(3, w, 19, \mathsf{add}, 0), (4, w, 3, \mathsf{del}, 0), (5, w, 1, \mathsf{del}, 1),$
  $(5, w, 4, \mathsf{del}, 2)$

The main difference to note is that in Level 3, the deletions are sorted according to their target level, and not simply according to the document identifiers. In general, in the new encoding, each level has a region in which it contains deletions for each level above it. Within that region, the deletions are in the same order as the additions appear in the corresponding upper level. This will enable us to execute an encrypted search very efficiently.

B. Detailed Protocol

In the previous section, we mentioned that given an appropriate token $\mathsf{token}_\ell(w)$, the server can decrypt the subtable at level ℓ corresponding to keyword w, denoted $\Gamma_\ell[w]$. This means:

1) The server can also look up an entry $\Gamma_\ell[w, \mathsf{op}, cnt]$ in $O(1)$ time.
2) Since for the same keyword w, the pairs (ℓ^*, id) appear in increasing order with respect to cnt in each level, the server can also perform a binary search on the field of (ℓ^*, id) (without knowing the respective cnt values). For example, given $\mathsf{token}_\ell(w)$, the server can decide whether an $(\ell^*, w, \mathsf{id}, \mathsf{op}, cnt)$ tuple exists in level ℓ in logarithmic time. This will be helpful later in the protocol.

Therefore, in this section, we will use the shorthand $\Gamma_\ell[w, \mathsf{op}, cnt]$ to denote a corresponding Lookup operation. We also explicitly write operations to the conceptual table Γ_ℓ for simplicity—but keep in mind that while we write w and operations to the Γ_ℓ table in the clear, the server performs these operations using the appropriate $\mathsf{token}_\ell(w)$ instead, *without actually seeing the search keyword* w.

Protocol $((\mathsf{st}', \mathcal{I}), \perp) \leftarrow \mathsf{Search}((\mathsf{st}, w), D)$

1) *Client:* Given a keyword w, the client computes a token for each level

$$\mathsf{tks} := \{\mathsf{token}_\ell := \mathsf{PRF}_{k_\ell}(h(w)) : \ell = 0, 1, \ldots, L\}.$$

The client sends the tokens tks to the server.
2) *Server:* Let $\mathcal{I} := \emptyset$. For $\ell \in \{L, L-1, \ldots, 0\}$ do:
   a) Find all tuples $(\ell, w, \mathsf{id}, \mathsf{add})$ in level ℓ, such that the corresponding delete operation $(\ell, w, \mathsf{id}, \mathsf{del})$ does not appear in levels $\ell' \leq \ell$.
   b) Set $\mathcal{I} := \mathcal{I} \cup \{\mathsf{id}\}$.

Return $\mathcal{I}$ to the client.

Fig. 6: The new and efficient search protocol.

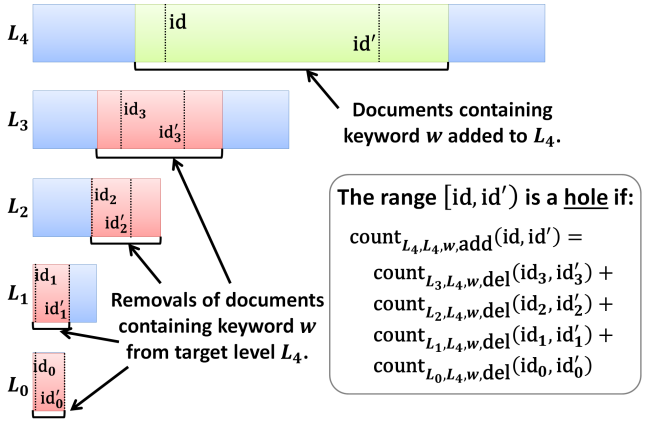

Fig. 7: The green region in level L_4 corresponds to all add entries of keyword w. The red regions in levels L_3, L_2, L_1, L_0 correspond to all del entries of keyword w that *were added* in level L_4 (that is why their target level is L_4). The goal of the $\mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id})$ algorithm is to find the largest green region (i.e., the largest hole) that contains entries that were all deleted in the lower levels. In the example above, $[\mathsf{id}, \mathsf{id}')$ is a hole because for all $x \in [\mathsf{id}, \mathsf{id}')$, there exists a $(L_4, w, x, \mathsf{del})$ entry in some lower level L_i for $i = 0, 1, 2, 3$. This condition is efficiently checked through the test in the box.

Finally, for any tuple $(\ell^*, w, \mathsf{id}, \mathsf{op}, cnt)$ stored at level ℓ we define

$$\mathsf{count}_{\ell,\ell^*,w,\mathsf{op}}(\mathsf{id}) = cnt\,.$$

For tuples $(\ell^*, w, \mathsf{id}_1, \mathsf{op}, cnt_1)$ and $(\ell^*, w, \mathsf{id}_2, \mathsf{op}, cnt_2)$ with $\mathsf{id}_2 \geq \mathsf{id}_1$ stored at level ℓ we also define

$$\mathsf{count}_{\ell,\ell^*,w,\mathsf{op}}(\mathsf{id}_2, \mathsf{id}_1) = cnt_2 - cnt_1\,.$$

New search protocol. In Figure 6 we present the new search protocol for a keyword w that operates on the data structure storing entries with the new encoding:

Note that Step 2a can take $O(N)$ time in the worst case. We now further show how to exploit the encoding

that we propose in the previous paragraphs and perform Step 2a a lot more efficiently, i.e., in $O(m \log^3 N)$ time. For that purpose we replace Step 2a and 2b with algorithm $\mathsf{ProcessLevel}(\ell, \mathsf{token}_\ell)$ of Figure 8.

The central idea of algorithm $\mathsf{ProcessLevel}(\ell, \mathsf{token}_\ell)$ is the following: Suppose a client has issued a token for keyword w and level ℓ. Instead of accessing all the add entries $(\ell, w, \mathsf{id}, \mathsf{add}, cnt)$ one-by-one by using successive values of the counter $cnt = 0, 1, 2, \ldots$ (as would happen in the search protocol of Section IV), the new protocol efficiently finds the new value of cnt that the server should use, so that to avoid accessing add entries that have been deleted in the lower levels. This set of entries are referred to as a *hole*. Finding the new value of cnt is performed in $O(\log^3 N)$ time by algorithm $\mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id})$ of Figure 8.

Algorithm $\mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id})$ matches a collection of successive "addition" identifiers appearing in level ℓ with a collection of successive "deletion" identifiers in lower levels. When a match is found (see $\mathsf{count}_{\ell,\ell,w,\mathsf{add}}(\mathsf{id}, \mathsf{id}') = \mathsf{DeletedSum}(\ell, \mathsf{id}, \mathsf{id}')$ in Figure 8), it can safely ignore the specific collection of addition identifiers and get the new value of the counter. A graphical representation of the $\mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id})$ algorithm in shown in Figure 7.

Time complexity. The complexity of the new search algorithm is $O(m \log^3 N)$, where m is the number of the documents that contain the keyword that is searched for and N is the current number of document-keyword pairs. This is because, for every one of the m documents in the search result, there are two cases:

- Either the algorithm encountered an add entry for the document and performed an unsuccessful binary search for the respective del entry in every level of the data structure, which clearly takes $O(\log^2 N)$ time;
- Or the the binary search for the respective del entry was successful. In this case the hole must be computed. To compute the hole, the algorithm performs a binary search on the level of the add entry, every step of which takes $O(\log^2 N)$ time, since the algorithm needs to go through all the levels below and perform a binary search within each level in order to compute the deleted sum. The overall task clearly takes $O(m \log^3 N)$ time.

Thus the running time of the new search algorithm is $O(m \log^3 N)$ in the worst case.

Achieving the $O(\min\{\alpha + \log N, m \log^3 N\})$ bound. In the introduction, we stated that the worst-case time complexity of the final search protocol is $O(\min\{\alpha + \log N, m \log^3 N\})$. To achieve this bound, we use a hybrid of the two search protocols that we described. Assume S_1 is the search protocol of Section IV (with $O(N)$ worst-case complexity) and S_2 is the search protocol of this section (with $O(m \log^3 N)$ worst-case complexity). The hybrid algorithm for searching for a keyword w is as follows:

1) Execute S_1 until $O(\log^2 N)$ addition entries for keyword w are encountered.
2) Through binary search at each level ℓ find the total number α of addition entries referring to keyword w.
3) If $\alpha = \omega(\log^3 N)$, then execute S_2; Else execute S_1.

The above algorithm runs in $O(\min\{\alpha + \log N, m \log^3 N\})$ worst-case time. To see that, we distinguish two cases. If the total number of addition entries $\alpha = O(\log^2 N)$, the algorithm will terminate at Step 1 in $O(\alpha + \log N)$ time. Otherwise, the algorithm will compute the exact number α in Step 2, which takes $O(\log^2 N)$ time. Then Step 3 computes the minimum. The reason we do not compute α from the very beginning is to avoid the $O(\log^2 N)$ cost.

Modifications to the Rebuild algorithm. We note here that we slightly need to modify the Rebuild algorithm presented in Section IV, to accommodate for the new encoding. Let ℓ be the level that is being rebuilt.

Specifically, we need to update the target level of all the entries that will be shuffled into the new level $\mathbf{T}_\ell$. This is easy to do with a sequential pass that takes place before the oblivious sorting. During this sequential pass, we set the target level of all entries (both add and del) to be ℓ **except** for the del entries whose current target level is $\ell' > \ell$ (we then accordingly re-encrypt the whole entry).

VI. Extensions and Optimizations

In this section we describe various extensions and optimizations for our scheme.

Supporting a passive server. Our algorithm can easily be transformed such that we can use a passive server, i.e., a server that only allows us to read and write data blocks and does not perform any computation. We can do this by having the client do the server's computation and accessing the data in storage that the server would have accessed. This has the negative impact of introducing more rounds into the protocol. For example, the search algorithm of our protocol is non-interactive because the server can perform binary search himself. In a passive server scenario, the binary search would have to be performed by the client, leading to a polylogarithmic number of rounds.

Achieving security in the malicious model. To make the sublinear construction work in the malicious model, the client stores a MAC of each data structure entry at the server, along with the entry itself. The MAC also includes the timestamp t and the *current* level ℓ of the entry.

During the execution of the Rebuild algorithm, when the client needs to read an entry from some level ℓ at the server, the server returns the entry along with its MAC and its timestamp t. The client can verify that the entry is correct, by recomputing the MAC and by comparing it against the returned MAC. When the client needs to write back an updated entry, the client uses the same timestamp but *the new level* for the computation of the MAC. This prevents replay attacks.

Algorithm $\mathsf{ProcessLevel}(\ell, \mathsf{token}_\ell)$

1) $cnt := 0$.
2) $(\ell, \mathsf{id}) := \Gamma_\ell[w, \mathsf{add}, cnt]$.
3) Repeat until (w, add, cnt) not in Γ_ℓ.
   - If $(\ell, w, \mathsf{id}, \mathsf{del})$ is not found in any lower levels:
     // through a binary search for each lower level
     - $\mathcal{I} := \mathcal{I} \cup \{\mathsf{id}\}$.
     - cnt++.
     - $(\ell, \mathsf{id}) := \Gamma_\ell[w, \mathsf{add}, cnt]$.
   - If $(\ell, w, \mathsf{id}, \mathsf{del})$ is found in some lower level (this is referred to as the start of a hole):
     - Call $cnt := \mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id}) + 1$.

Algorithm $\mathsf{SkipHole}(\ell, \mathsf{token}_\ell, \mathsf{id})$

1) Through binary search, compute the maximum identifier $\mathsf{id}' > \mathsf{id}$ in level ℓ such that
$$\mathsf{count}_{\ell,\ell,w,\mathsf{add}}(\mathsf{id}, \mathsf{id}') = \mathsf{DeletedSum}(\ell, \mathsf{id}, \mathsf{id}') .$$
2) Return the corresponding cnt value for id'.

Algorithm $\mathsf{DeletedSum}(\ell, \mathsf{id}, \mathsf{id}')$

1) $sum := 0$.
2) For each level $\ell' < \ell$:
   - Find the region $[(\ell, \mathsf{id}_x), (\ell, \mathsf{id}_y)]$ that falls within the range $[(\ell, \mathsf{id}), (\ell, \mathsf{id}')]$ (through binary search), and compute
$$r := \mathsf{count}_{\ell',\ell,w,\mathsf{del}}(\mathsf{id}_y, \mathsf{id}_x) .$$
   - $sum := sum + r$.
3) Return sum.

Fig. 8: Algorithms for processing the level efficiently.

For the verification of a search operation, the server needs to prove to the client that it has (1) returned the correct results, and (2) returned all of the results. To achieve these guarantees, recall that when the client performs a search for a keyword w, it sends search tokens that allows the server to unlock a set of entries in each level that correspond to the keyword w. To protect against a possibly malicious server, we require that the server send the following to the client for each level ℓ.

- Each add entry in level ℓ which has not been deleted (i.e., via a del entry in a lower level). Also, a MAC for each such entry along with its timestamp.
- A proof of each hole. Recall that a hole is a set of consecutive add entries each of which have been deleted (i.e., via a del entry in a lower level). The proof of the hole consists of the del entries at the edge of each deletion region in the lower levels, along with a MAC for each such del entry.

After receiving this information, the client can reconstruct the sums of each deletion region (by using the counters) below a certain level and verify that the sum of those sums is equal to the sum of entries in the hole.

Removing the random oracle. Our scheme uses the random oracle. We can however replace the random oracle by a PRF: For the encoding of the entry, the client computes

$$\mathsf{hkey} := \mathsf{PRF}_{k_\ell}(w||0||\mathsf{op}||cnt)$$

and

$$c_1 := \mathsf{id} \oplus \mathsf{PRF}_{k_\ell}(w||1||\mathsf{op}||cnt) .$$

When searching for a keyword w, the client now needs to provide more tokens: For each level ℓ, it gives the PRF outputs $\mathsf{PRF}_{k_\ell}(w||0||\mathsf{op}||cnt)$ and $\mathsf{PRF}_{k_\ell}(w||1||\mathsf{op}||cnt)$ for $cnt = 0, 1, \ldots$, instead of $O(\log N)$ tokens that would enable the server to compute such values. However the drawback of this method is that the client needs to do more computation now and also the client-server communication increases.

Resizing the data structure. The example at the beginning of Section V requires to store a data structure of size $O(N)$, although the actual number of documents, after the deletions have taken place, is $O(1)$. This causes a blow-up in the space of our data structure. We however note that this problem can be easily addressed: Whenever the number of deleted entries equals $N/2$ (where N is the current total number of entries in the data structure), the protocol rebuilds the data structure from scratch, eliminating duplicate entries. This assures that we always use $O(N)$ space, where N is the *actual* number of entries.

Since the time to rebuild the data structure is $O(N \log N)$, the bandwidth for the rebuild is $O(N)$ and the rebuild happens every at least N operations, it follows that the asymptotic worst-case update time and the asymptotic worst-case update bandwidth is not influenced.

VII. Experimental Results

We implemented our sublinear construction (Section V) in C# consisting of about 4000 lines of code. Our experiments were performed on Amazon EC2 on a cr1.8xlarge instance running Windows Server 2008 R2 containing two Intel Xeon E5-2670 2.6GHz processors and 244GB of RAM. All experiments were performed on the same machine using inter-thread RPC with simulated network round-trip latencies ranging from 25ms to 100ms.

Number of trials. Each data point in the graphs is an average of 10 trials. We omitted error bars because the variance was low enough that error bars are too small.

Client storage. In all of our experiments, we configured our implementation to use less than 210 MB of client storage at all times. All of the storage except for the level keys is transient and is only needed to perform updates. The transient storage can always be discarded if necessary and level rebuilding can be restarted. Even many mobile clients today can afford such client storage. It is also a tunable parameter which can be adjusted in practice.

Dataset. The search performance of a query depends only on the number of results of matching this query and the number of keyword-document pairs of the database. It does not depend on any other parameters, such as the contents of the database (as should be the case to ensure privacy). Hence, we describe the performance of our system in terms of dataset content-independent metrics. We measure our performance based on the size of the database, number of results per query, and network latency. In our experiments, the dataset and server data structures are stored in RAM on our machine.

Deamortization. Our implementation uses the deamortized Rebuild (not SimpleRebuild) algorithm and is constantly "rebuilding", to spread the rebuilding work over time and avoid a large worst-case cost. Searches can occur at any time and the client and server do not have to wait for a rebuild to complete before performing a search.

A. Adding and Removing Documents

We now evaluate the performance of update operations (i.e., adding and removing documents) in our scheme.

Update throughput. Because of our hierarchical data structure, the update time is proportional to $O(\log^2 N)$ where N is the current size of the database (i.e., the number of document-keyword pairs already in the database).

Figure 9 shows the maximum sustainable rate at which our implementation can perform updates on the database.

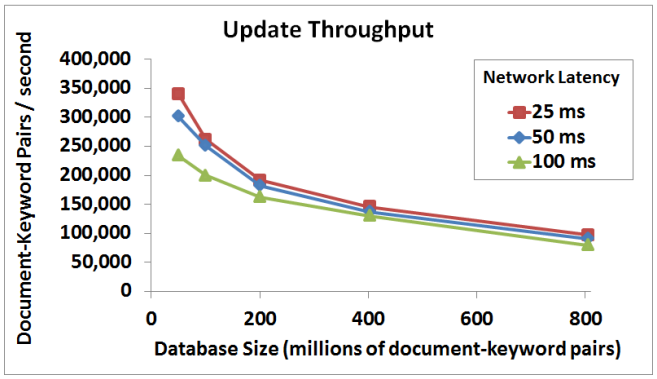

Fig. 9: **Update throughput of our sublinear construction.** The update time is specified in keyword-document pairs per second. For example, adding or removing a document with 100 unique keywords results in 100 document-keyword pair updates.

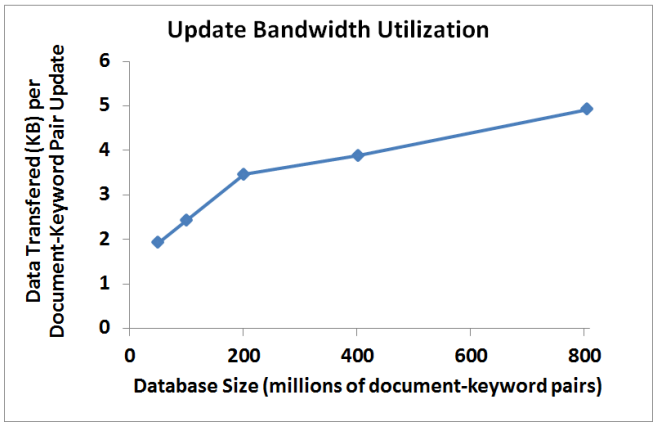

Fig. 10: **Update Bandwidth.** The bandwidth used to add or remove a document-keyword pair from the database. Typical network latencies have a small effect on throughput.

We are able to add or delete document-keyword pairs at the rate from 100,000 pairs per second for an 800 million pair database to over 300,000 pairs per second for a 50 million pair database. Note that adding and removing a pair results in the same kind of operation so the performance is the same.

The rate at which documents can be added and removed depends on the size of the documents. For example, adding or removing a document with 100 unique keywords will result in 100 document-keyword pair updates.

In the above throughput experiment, we assume a setup with sufficient client-server bandwidth. The overhead is therefore dominated by the client-side computation cost of performing decryption and encryption, and sorting of the entries. Below we measure how much bandwidth is required for update operations.

Update bandwidth cost. Each update of a document-

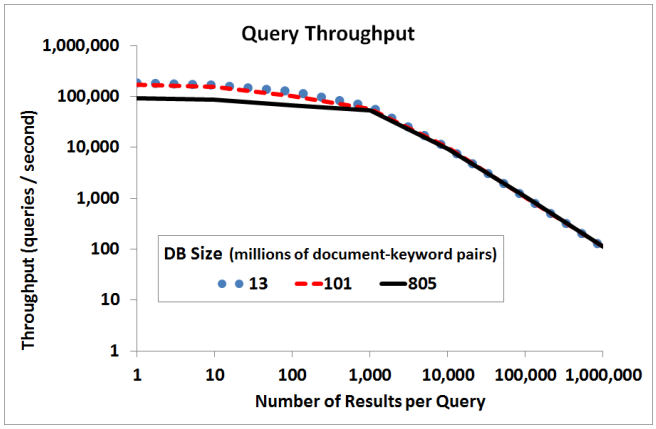

Fig. 11: **Query throughput of our sublinear construction.** Our system is able to execute queries at a rate of up to 100,000 queries/second. For large result sets, the query throughput depends inverse proportionally to the number of results (note that both axis are log-scale).

keyword pair, triggers $O(\log^2 N)$ work on average for rebuilding some of the existing levels in the database as described in Sections IV and V. Even though the average computation per update is $O(\log^2 N)$, the average amount of bandwidth consumed per update is actually $O(\log N)$ as mentioned in Section IV-D. Figure 10 shows our measurements for the average update bandwidth cost for several databases sizes. Note that since we use standard de-amortization techniques [15], [16], [37], we are able to perform a partial level rebuilds so that our worst case computation and bandwidth update costs are within a constant factor of the average costs.

Effect of network latency. We tested the throughput under 25ms, 50ms, and 100ms simulated network round-trip latencies. For large databases (e.g., larger than 400 million keyword-document pairs), the throughput was only slightly affected by network latency. For smaller databases (e.g., 20 million pairs) the throughput decreased by about 30 percent as a result of increasing the latency from 25ms to 100ms.

B. Searching

In Figure 11, we measure the rate at which our database can sustain different sized queries. For queries with less than 1,000 results, we can execute them at about 100,000 queries per second. The rate is fairly constant for these small queries because the search RPC cost dominates the running time.

For larger queries, our throughput is inversely proportional to the number of results. We can achieve a rate of 9,000 queries per second each with 10,000 results per query and over 100 queries per second with 1 million results per query.

The search time for larger queries is mostly unaffected by the database size because the work done is proportional to the number of entries accessed regardless of how they are distributed within the levels of the data structure. Because every level must be accessed at least once, smaller queries are slightly less efficient on larger databases as can be seen in the Figure 11.

Number of Keyword-Document Pairs	DB Size (GB)
50,331,648	20
100,663,296	42
201,326,592	82
402,653,184	130
805,306,368	202

TABLE I: **Database size.** Measured while performing updates. The ratio between the number of keyword-document pairs and the size varies slightly as our de-amortized updater is running.

C. Database Size

As shown in Table I, our database sizes varied form 20GB to 202GB for databases with 50 million to 805 million document-keyword pairs. The databases stores about 250 to 400 bytes per keyword-document pair. The number of bytes varies throughout the lifetime of the system due to the temporary storage allocated and deallocated for rebuilding levels.

D. RAM vs. Disk

One interesting question to ask is that if we were to support a desired throughput, whether RAM or disk will be more economical as the storage medium for large-scale searchable encryption systems like ours.

RAM is more economical. We found that in scenarios where queries are frequent, a RAM based approach is several orders of magnitude more economical than using disks for the same query throughput. For example, we show that using 244 GB RAM, we can sustain a search throughput of $100,000$ queries per second.

We performed a back-of-the-envelope calculation on how many disks are needed to sustain the same throughput, partly based on the performance numbers reported by Cash et al. [5] who implemented a static, conjunctive searchable encryption scheme on enterprise-grade rotational hard-drives. They are able to process about 1 query (with 10,000 results) per second with 6 enterprise-grade disks [1]. In fact, in order to get 1 query per second performance they use a document-grouping trick which leaks additional information about keyword frequencies in the database [1], [5]. Without resorting to this extra leakage, one query would take about 10 seconds to execute. In contrast, our RAM based implementation is able to achieve over 9,000 queries per second (90,000 times faster) without this extra leakage.

Cash et al.'s scheme is a conjunctive search scheme, however, their disk performance is still indicative, since they report that their performance is disk I/O bound for searches. In particular, each query makes random disk seeks proportional to the number of documents that match their

s-term (i.e., the size of the result set if a single keyword is searched). For large result sets, our schemes would require a similar (in the ballpark) number of disk I/O operations per search query, we anticipate similar search performance if we implemented it on disk.

We can now do a back-of-the-envelope calculation to see how many parallel disks are required to sustain the throughput of our RAM-based implementation—in fact, this would require about 500K (in the ballpark) enterprise-grade disks. Hence using RAM when queries are frequent is more economical.

Cash et al. also report their performance results under RAM storage. Under a single keyword search, and a result set of 1000, their RAM response time is 0.1s. For our scheme, at 1000-sized result set, our throughput is more than $50,000$ queries per second. Our system has a maximum degree of parallelism at 32, with 16 cores (with hyper-threading). This means that absent any queuing and network delay, the (computational) response-time of our scheme is under 1ms.

Practical considerations. One potential concern with storing the database in RAM is that power loss or system failure could lead to data loss (because RAM is volatile memory). New systems such as RAMCloud [26] have been proposed to offer both persistent storage as well as the fast random access of RAM. We can also address this issue as follows. Our level rebuilding algorithm results in a highly sequential workload when reading and writing entries in the levels (including the oblivious sorts). Therefore we could efficiently perform the updates on disk (with few seeks) while still caching an in-memory copy of the database for processing search queries (which would require many seeks if done on disk).

VIII. Proof of Basic Construction

We now build a simulator $\mathcal{S}$, which given the leakages $\mathsf{leak}_u(\mathsf{upd})$ $\mathsf{leak}_s(w_i)$ (as defined in Section III-B) for each update and search operation respectively, simulates the interactions with a real-world, semi-honest server $\mathcal{A}$.

A. Simulation of the Update Protocol

1) SimpleRebuild *Case:* The simulator $\mathcal{S}$ learns only the leakage $\mathsf{leak}_u(\mathsf{upd})$ for each update upd, including the identifier of the added/deleted document, the type of operation, the number $|\mathbf{w}|$ of keywords of the added/deleted document as well as the time that the operation was performed.

For $i \in \{1, 2, \ldots, |\mathbf{w}|\}$, the simulator simulates the Rebuild protocol $|\mathbf{w}|$ number of times.

For each Rebuild protocol, suppose level ℓ is being rebuilt. The simulator simply creates a "random" level $\mathbf{T}_\ell$ as follows: for each entry in $\mathbf{T}_\ell$, the simulator creates a "random" encoded $\mathsf{entry} := (\mathsf{hkey}, c_1, c_2)$. Specifically, the hkey and c_1 terms of the entry will be generated at random. The c_2 term is a semantically-secure ciphertext, and therefore can be simulated by simply encrypting the **0** string.

2) Rebuild *Case:* We now show that if there exists a simulator that can simulate the SimpleRebuild case, then we can build a simulator to simulate the Rebuild protocol. We recall that in the oblivious sorting algorithm (the only difference between SimpleRebuild and Rebuild), the client downloads a small subsets of the entries ($O(N^\alpha)$), sorts them locally, and uploads them back to the server (possibly to different locations). After several rounds of sorting subsets of entires, the entire level becomes sorted. Therefore the simulator for the Rebuild case calls the SimpleRebuild simulator to simulate the processing of each $O(N^\alpha)$-sized subset. By the existence of a simulator for the SimpleRebuild case and by the obliviousness of the sorting algorithm, it follows the new simulator can successfully simulate the Rebuild protocol.

B. Simulation of the Search Protocol

For the Search protocol for word w_i, the simulator learns leakage $\mathsf{leak}_s(w_i)$ as defined in Section III-B, that includes the number of matching documents $\{\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_r\}$, as well as when keyword w_i was searched previously.

For each filled level ℓ, the simulator computes a random token_ℓ and sends it to the real-world adversary $\mathcal{A}$—note that due to the pseudorandomness of the PRF function, the adversary cannot distinguish a random token from a pseudorandom one.

For the server to be able answer the search query, we need to program the random oracle as below.

Programming the random oracle. First, note that the probability that the adversary queries the random oracle for token_ℓ before token_ℓ is given to the adversary is negligible.

Second, for all random oracle queries that have been made before, the simulator just returns the same answer.

After the adversary gets token_ℓ for a specific filled level ℓ, the adversary can query the random oracle for $\mathsf{H}_{\mathsf{token}_\ell}(0||\mathsf{op}||cnt)$ and $\mathsf{H}_{\mathsf{token}_\ell}(1||\mathsf{op}||cnt)$ for various cnt values.

Now, from the search leakage, the simulator knows the matching set of document identifiers, the time that each document was added and removed, and the number of keywords in each document.

Given this information, for each matching document id, the simulator must compute which level $\mathbf{T}_\ell$ the tuple $(w, \mathsf{op}, \mathsf{id})$ will appear, for both add and del operations. Specifically, since, for all update operations it knows the order of their execution and the number of unique keywords contained in the document of every update, it can replay these updates and figure out the levels that specific tuples $(w, \mathsf{id}, \mathsf{op})$ occupy for the **same** $(\mathsf{id}, \mathsf{op})$ field. Specifically, we have two cases:

1) All tuples $(w, \mathsf{id}, \mathsf{op})$ with the **same** $(\mathsf{id}, \mathsf{op})$ field occupy one level $\mathbf{T}_\ell$. Then, the simulator outputs level $\mathbf{T}_\ell$ as the level that id (contained in the search output) is stored.

2) All tuples $(w, \mathsf{id}, \mathsf{op})$ with the **same** $(\mathsf{id}, \mathsf{op})$ field span multiple consecutive levels $\mathbf{T}_j, \mathbf{T}_{j+1}, \ldots, \mathbf{T}_{j+k}$ such that there are x_i occurrences of such tuples in $\mathbf{T}_i$ for $i = j, \ldots, j+k$. Then, the simulator outputs level $\mathbf{T}_r$ $(j \leq r \leq j+k)$ as the level that id (contained in the search output) is stored with probability

$$p_r = \frac{x_r}{\sum_{i=j}^{j+k} x_i}\,.$$

Note that the level computed above is indistinguishable from the level of the real execution because all tuples $(w, \mathsf{id}, \mathsf{op})$ with the **same** $(\mathsf{id}, \mathsf{op})$ are added with **random** order during the update protocol, and therefore they can end up in the different levels $\mathbf{T}_r$ with the probabilities p_r shown above.

Therefore, the simulator can determine that for a specific level ℓ, the matching entries are:

$$\left[\begin{array}{l} (\mathsf{id}, \mathsf{add}, 0), (\mathsf{id}, \mathsf{add}, 1), \ldots, (\mathsf{id}, \mathsf{add}, cnt_{\mathsf{add}}), \\ (\mathsf{id}, \mathsf{del}, 0), (\mathsf{id}, \mathsf{del}, 1), \ldots, (\mathsf{id}, \mathsf{del}, cnt_{\mathsf{del}}) \end{array}\right]$$

In other words, for each (op, cnt) pair, the simulator knows the appropriate document id.

Therefore, if the adversary queries $\mathsf{H}_{\mathsf{token}_\ell}(b||\mathsf{op}||cnt)$ for some $cnt > cnt_{\mathsf{op}}$, the simulator just returns random bitstrings.

Suppose the adversary queries $\mathsf{H}_{\mathsf{token}_\ell}(b||\mathsf{op}||cnt)$ for some $0 \leq cnt \leq cnt_{\mathsf{op}}$. The simulator knows the corresponding id for (op, cnt) in level $\mathbf{T}_\ell$. The simulator picks a random, "unassigned" entry in level $\mathbf{T}_\ell$—if it hasn't already done so for the tuple (w, op, cnt). This entry now is marked as "assigned", and will now be the entry associated with the tuple $(w, \mathsf{op}, \mathsf{id}, cnt)$. Suppose this selected entry is $\mathsf{entry} := (\mathsf{hkey}, c_1, c_2)$. The simulator returns hkey for the random oracle query $\mathsf{H}_{\mathsf{token}_\ell}(0||\mathsf{op}||cnt)$, and $\mathsf{id} \oplus c_1$ for the random oracle query $\mathsf{H}_{\mathsf{token}_\ell}(1||\mathsf{op}||cnt)$.

IX. Proof of the Full Construction

Given the proof of the basic construction, the proof for the full scheme is straightforward.

Note that the only difference between the full and basic scheme is the following:

- In the full scheme, we add a target level ℓ^* which is encoded in the same way as the document identifier id.
- The full scheme releases the same token to the server during each search operation, and no additional information. The full scheme is more efficient only because the server implements a more efficient algorithm to locate the desired information—in terms of information revealed to the server, there is no difference from the basic scheme.
- The rebuild algorithm has to additionally compute the new target levels.

Therefore, the simulation of the full scheme is almost the same as the basic scheme. The only difference is that the simulator needs to take the target level ℓ^* into account when answering random oracle queries (i.e., the oracle does not output only document identifiers).

Note again, that the chance that the adversary $\mathcal{A}$ makes a random oracle query on some token before token is given to $\mathcal{A}$ is negligible. Therefore, we can assume that the adversary only makes a random oracle on some token after token is given the $\mathcal{A}$ in a search query. At this moment, the matching set of documents (both add and del entries) for the searched keyword w is revealed to the simulator. The simulator also knows when each document is added and/or deleted, and how many keywords are in each document. Therefore, at this moment, the simulator can emulate the $(w, \mathsf{id}, \mathsf{op})$ tuples in each level $\mathbf{T}_\ell$—note that the simulator can do this after the set of documents matching w is revealed to the simulator. Therefore, the simulator can compute the target levels related to the searched keyword w for every level, $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, and for all matching document identifiers (for existing or removed documents).

Namely, given the identifiers of the documents matching a keyword w (both add and delete entries)—which are included in the leakage, the simulator can directly compute the correct target levels and return those in the output of the random oracle.

References

[1] Personal communication with Hugo Krawczyk, Michael Steiner, and Marcel Rosu.

[2] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.

[3] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[4] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, pages 253–273, 2011.

[5] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

[6] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *CCS*, 2013.

[7] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[8] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on*

Computer and Communications Security (CCS '06), pages 79–88. ACM, 2006.

[10] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[11] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[12] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.

[13] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[14] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[15] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[16] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious ram simulations. *CoRR*, abs/1007.1259, 2010.

[17] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Cloud Computing Security Workshop (CCSW)*, 2011.

[19] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[20] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography (FC)*, 2013.

[21] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security*, pages 965–976, 2012.

[22] K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In *Financial Cryptography*, pages 285–298, 2012.

[23] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[24] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1990.

[25] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[27] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[28] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In *TCC*, pages 457–473, 2009.

[29] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, 2007.

[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[31] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2000. IEEE Computer Society.

[32] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, pages 253–267, 2013.

[33] E. Stefanov, E. Shi, and C. Papamanthou. Dynamic proofs of retrievability without oblivious RAM. In *CCS*, 2013.

[34] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[35] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

[36] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management*, pages 87–100, 2010.

[37] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[38] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.