Presenter: Gungor Basa

Practical Dynamic Searchable Encryption with Small Leakage

Emil Stefanov, Charalampos Papamanthou, Elaine Shi
Content

❖ Introduction
❖ Data Structures
  ❖ Server Side
  ❖ Client Side
❖ Simple Algorithm
❖ Advanced Algorithm
❖ Performance
❖ Conclusion
Introduction
Cloud Computing

Client Outsources his data
Searches and updates

**Question:** Why should I trust my cloud provider?
**Answer:** You should not

**Question:** Can Encryption solve my trust issue?
**Let me think about it...**
Encryption

❖ Great way to hide data
❖ Can we search inside of the encrypted data without telling anything to the server (Cloud Provider)? NO
❖ We need something new
Searchable Encryption

Searchable Encryption enables a client to encrypt her document collection in a way that keyword search queries can be executed the encrypted data via the appropriate “keyword tokens”.

There are 2 kinds of SSE; Dynamic and Static. Our scheme is a Dynamic SSE.
Contribution

- Paper presents a method that achieves
  - **Small Leakage**
  - **Efficiency** (Update and Search sublunar time in worst case)
- Until the time paper was written, **None of the DSSE Schemes** can achieve both
Leakage

- All existing DSE schemes leak;
  - **Search Pattern:** Hashes of keywords we are searching for
  - **Access Pattern:** Matching document identifiers
  - **Size Pattern:** Current size of the index (N)

**Forward Privacy:** If we search for a keyword “w” and later add a new document containing keyword “w”, server does not learn that the new document has a keyword we searched for in the past

**Backward Privacy:** Queries cannot be executed over deleted documents. (NO)
Leakage

- Apart from the previous leakage, this method also leaks the document identifiers that were deleted in the past and match the document (No backward privacy)
- It still leaks way less than other methods

Forward Privacy: If we search for a keyword “w” and later add a new document containing keyword “w”, server does not learn that the new document has a keyword we searched for in the past

Backward Privacy: Queries cannot be executed over deleted documents. (NO)
Efficiency

- **Worst Case Search Complexity**: 
  - $O(\min \alpha + \log_2 N, m \cdot \log_2^3 N)$
- **Worst Case Update Complexity**
  - $O(k \cdot \log_2^2 N)$
- **Space**
  - $O(N)$

N: #of document-keyword pairs
m: Number of documents containing the keyword we are searching for
alpha: number of times this keyword was historically added to the document
k: # of unique keywords contained in the document of update
A dynamic searchable encryption scheme is a suite of three protocols:

1. Setup
2. Search
3. Update
Setup

- \((st, D) \leftarrow \text{Setup}(N, \ldots)\)
- **Client** counts the number of document-keyword pairs and send this number to the server
- **Server** gets the number of document-keyword pairs and construct a data structure based on N
- **Output** client gets his secret state, server has its data structure
Search

- \((st', I) <- \text{Search}((st, w), D)\)
- **Client** inputs \(st\) and a keyword
- **Server** inputs the data structure
- **Output** client get possible updated \(st'\) and the set of documents identifiers that contains the keyword \(w\)
Update

❖ \((st', D') \leftarrow ((st, upd), D)\)
❖ **Client** inputs the st and update operation \((upd := (\text{add, id, w}) \text{ or } (\text{del, id, w}))\)
❖ **Server** inputs the data structure
❖ **Output** client gets new updated st, and server gets new updated data structure \(D'\)
Data Structures

❖ Server Side
❖ Client Side
Server Side Data Structure

- Stores document-keyword pairs in a hierarchical structure of logarithmic levels
- For given $N$, data structure contains $\log_2 N + 1$ levels
- For each level it stores up to $2^l$ entries
Server Side Data Structure

\[ l = 4 \quad 2^4 = 16 \]

\[ l = 3 \quad 2^3 = 8 \]

\[ l = 2 \quad 2^2 = 4 \]

\[ l = 1 \quad 2^1 = 2 \]

\[ l = 0 \quad 2^0 = 1 \]
Adding entries is not straightforward.

With a small modification to the straightforward way, we can have better performance.

Let's say \( N = 16 \) and see how the data structure works in addition.
Add 1st Entry
Add 2nd Entry
Add 3rd Entry
Add 4th Entry
Add 5th Entry
<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry 1</td>
<td></td>
</tr>
<tr>
<td>Entry 2</td>
<td></td>
</tr>
<tr>
<td>Entry 3</td>
<td></td>
</tr>
<tr>
<td>Entry 4</td>
<td></td>
</tr>
<tr>
<td>Entry 5</td>
<td></td>
</tr>
<tr>
<td>Entry 6</td>
<td></td>
</tr>
</tbody>
</table>
Add 7th Entry
Add 8th Entry
Add 9th Entry
Add 10th Entry
Add 11th Entry
Add 12th Entry

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Add 14th Element
Add 15th Element
Add 16th Element (Last)
Token Structure

- What are we actually storing on the server? Encrypted File + Token
  - **Add Token** = (w, doc_id, add, counter)
  - **Delete Token** = (w, doc_id, del, counter)
- Not in a plain text way?? To be able to understand this better we will examine the basic algorithm.
Client Side Data Storage

- Nothing fancy about it
- **Client** needs to store private encryption key + key for each level
- **Array of keys** would work just fine
Basic Algorithm

- With the basic algorithm we cannot achieve mentioned efficiency, however it is good to understand for the approach.
- Let’s say we have a document which has a keyword, let’s see how algorithm works.
Basic Algorithm

❖ First, initial setup phase (Constructs Data Structure)
❖ EncodeEntry (Create Tuple)
I got this question from prof before, but it does not mention in the paper. However, I realized that if we don't xor we reveal all keywords for a document.
Update

Protocol \((st', D') \leftarrow Update((st, upd), D)\)

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

For \(w \in w\) in random order do:

- If \(T_0\) is empty, select a fresh key \(k_0\) and set \(T_0 := \text{EncodeEntry}_{esk,k_0}(w, \text{id}, \text{op}, cnt = 0)\).
- Else, let \(T_\ell\) denote the first empty level:
  - Call \(\text{SimpleRebuild}(\ell, (w, \text{id}, \text{op}))\).
  - (or \(\text{Rebuild}(\ell, (w, \text{id}, \text{op}))\).

\(O(\log^2 N)\)
Protocol SimpleRebuild($\ell$, ($w$, id, op))
(Assuming $O(N)$ client working storage)

1) Client creates local buffer $B = (w, id, op, cnt = 0)$.
2) For each entry $=(hkey, c_1, c_2) \in T_0 \cup T_1 \cup \ldots \cup T_{\ell-1}$:
   Let $(w, id, op, cnt) := \text{Decrypt}(c_2)$.
   Let $B := B \cup (w, id, op, cnt)$.
   // Client: download and decrypt all entries, store in local $B$.
3) Sort $B$ based on lexicographical sorting key $(w, id, op)$.
   // All entries with the same keyword now appear sequentially.
4) For each $e := (w, id, op, cnt') \in B$ (in sorted order):
   • If $e$ marks the start of a new word $w$, for an operation $op \in \{\text{add}, \text{del}\}$, then set $cnt_{op,w} := 0$ and update $e := (w, id, op, 0)$ in $B$.
   • If $e$ and its adjacent entry are add and del operations for the same $(w, id)$ pair, suppress the entries by updating both entries with $\bot$.
   • Else, update $e := (w, id, op, cnt_{op,w}++)$ in $B$.
5) Select a fresh new level key $k_\ell$.
   $T_\ell := \{\text{EncodeEntry}(e_{k_{\ell}}, (entry))\}_{\text{entry} \in B}$.
   // Dummy entries marked $\bot$ are also encoded as part of $T_\ell$.
   Upload $T_\ell$ to the server in the order of increasing hkey. Empty all the old levels $T_0, T_1, \ldots, T_{\ell-1}$.

$O(N\log N)$
Basic Structure

- Until this point, we successfully add an entry to the server side data structure
- What about searching?
Algorithm $\text{Lookup}(\text{token}, \text{op}, \text{cnt})$

1. $\text{hkey} := H_{\text{token}}(0||\text{op}||\text{cnt}).$
2. If $\text{hkey} \notin T_{\ell}$, output $\bot$.
3. Else, output $\text{id} := T_{\ell}[\text{hkey}].c_1 \oplus H_{\text{token}}(1||\text{op}||\text{cnt}).$

Returns $\text{id}$
Search

**Protocol** $(\text{st}', \mathcal{I}), \bot) \leftarrow \text{Search}(\text{st}, w, D)$

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

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

   The client sends the tokens $\text{tks}$ 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:
     
     $id := \text{Lookup}(\text{token}_\ell, \text{add}, cnt)$.
     
     $\mathcal{I} := \mathcal{I} \cup \{id\}$.

   - For $cnt := 0, 1, 2, 3, \ldots$ until not found:
     
     $id := \text{Lookup}(\text{token}_\ell, \text{del}, cnt)$.
     
     $\mathcal{I} := \mathcal{I} - \{id\}$.

   Return $\mathcal{I}$ to the client.
Basic Structure Diagram
Advanced Algorithm

- We think client has small local storage
- In this scenario most straightforward approach is treat server as an ORAM, it has too many disadvantages. So we use a reminiscent method

We cannot use backward security. However, ORAM is backward secure. We use o-sort
Protocol Rebuild($\ell, (w, id, op))$
(Assuming $O(N^0)$ client working storage, $0 < \alpha < 1$)

1) Let entry$^*$ := EncodeEntry$_{\text{ek}, h_0}(w, id, op, cnt = 0)$.
   Let $\mathcal{B} := \{\text{entry}$$^*$\} \cup \mathcal{T}_0 \cup \mathcal{T}_1 \cup \ldots \cup \mathcal{T}_{\ell-1}$.
3) $\mathcal{B} := \text{o-sort}(\mathcal{B})$, based on the lexicographical sorting key $(w, id, op)$.
   
   // Now, all entries with the same key appear sequentially.
4) For each entry $e := \text{Encrypt}_{\text{ek}}((w, id, op, cnt^*), \mathcal{B})(s)$ in $\mathcal{B}$ (in sorted order):
   
   - If $e$ marks the start of a new word $w$, for an operation $op \in \{\text{add, del}\}$, then set $\text{cnt}_{op, w} := 0$ and update $e := \text{Encrypt}_{\text{ek}}((w, id, op, 0), \mathcal{B})$.
   - If $e$ and its adjacent entry are add and del operations for the same $(w, id)$ pair, suppress the entries by updating both entries with $\text{Encrypt}_{\text{ek}}(\bot)$.
   - Else, update $e := \text{Encrypt}_{\text{ek}}((w, id, op, \text{cnt}_{op, w}++), \mathcal{B})$.
5) Randomly permute $\mathcal{B} := \text{o-sort}(\mathcal{B})$, based on $h_0$.
6) Select a new level key $k_{\ell}$.
   For each entry $e \in \mathcal{B}$:
   
   $(w, id, op, cnt) := \text{Decrypt}_{\text{ek}}(e)$
   Add $\text{EncodeEntry}_{\text{ek}, k_{\ell}}(w, id, op, cnt)$ to $\mathcal{T}_{\ell}$.
O-Sort

- It is based on a merge sort algorithm that merges 2 sorted halves
- Inherits from a sorting network, so it is parallelisable
- Number of comparators $O(n \log^2 n)$

A sorting network is a mathematical model of a network of wires and comparator models that is used to sort sequence of numbers.

Client reads and writes from/to servers memory

Everytime an entry is uploaded back to server, it is re-encrypted with a different nonce so the server cannot link the positions of the reordered entries to their original intermediate positions before and during sorting

Server cannot learn anything about the entries by observing o-sort

$O(N \log N)$ client side computation

Based on Odd-Even Sort (External Memory Sort)
Data Oblivious Sorting (O-Sort)

❖ The algorithm is data-oblivious if the sequence of I/O that it performs is independent of the values of the data it is processing.
❖ O-Sort is a kind of Merge-Sort
❖ It allows us to sort the data piece by piece in the client side ($O(n^\alpha)$)
❖ It may sort the same thing more than once

ORAM, allow a client to conceal its access pattern to the remote storage by continuously shuffling and re-encrypting data as they are accessed.
Old search operation is not $O(m\log^3 N)$, let's make it

Especially when $k = N$ all of the document contain a keyword $w$
Improving Search

- In the new structure, for each tuple \((w, id, op, cnt)\), stored at the level \(l\) we will also store \(l^*\) such that
  - If \(op = \text{add}\), entry will be stored i.e. \(l^* = l\)
  - If \(op = \text{del}\), then \(l^*\) is the level of data structure that respective addition tuple \((w, id, \text{add}, cnt)\) is stored i.e. \(l^* > l\)
- Then when we are sorting we sort it based on \((l^*, w, id, op, cnt)\)
New Encoding

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

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

\[(\ell^*, \text{w}, \text{id}, \text{op}, \text{cnt})\]

Server will decrypt
Improving Search

Let's say we have documents 1, 4, 13, 3 containing keyword w were added, then documents 1, 3, 4 are deleted and finally document 19 containing keyword was added.

Main difference between 2 methods is that in level3. The deletions are sorted according to their target level, not simply according to the id. In general each level has a region in which it contains deletions for each level above it. This will help us to nullify values with another method.

With this method server can decide whether an (l*, w, id, op, cant) tuple exists in level l in logarithmic time because it can perform binary search.
Improving Search

Documents containing keyword $w$ added to $L_4$.

The range $[id, id']$ is a hole if:

$$
\text{count}_{L_4, L_4, w, \text{add}}(id, id') = \\
\text{count}_{L_3, L_4, w, \text{del}}(id_3, id_3') + \\
\text{count}_{L_2, L_4, w, \text{del}}(id_2, id_2') + \\
\text{count}_{L_1, L_4, w, \text{del}}(id_1, id_1') + \\
\text{count}_{L_0, L_4, w, \text{del}}(id_0, id_0')
$$
Note that 2a can take $O(N)$ in the worst case

Remember I was talking about calculating and deleting holes, now it is time to do it :)

---

**Protocol**

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

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

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

   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 $(\ell, w, \text{id, add})$ in level $\ell$, such that the corresponding delete operation $(\ell, w, \text{id, del})$ does not appear in levels $\ell' \leq \ell$.

   b) Set $\mathcal{I} := \mathcal{I} \cup \{\text{id}\}$.

   Return $\mathcal{I}$ to the client.
To be able to make our search algorithm more efficient, we will change 2a and 2b with algorithm

ProcessLevel(l, token_l)

Instead of $O(N)$, it will perform the same task in $O(m \log^3 N)$

The central idea of ProcessLevel is following;
Suppose a client has issued a token for keyword $w$ and level $l$. Instead of accessing all the add entries $(l, w, id, add, cnt)$ one-by-one using successive values of the counter $cnt = 1, 2, 3, \ldots, n$, the new protocol efficiently finds the new value of $cnt$ that the server should use.
All this skipHole part aims to find the next cnt value without calculating all cnt = 1,2,3, n
Improving Search - SkipHole

**Algorithm SkipHole(ℓ, token, id)**
1) Through binary search, compute the maximum identifier id' > id in level ℓ such that
\[
\text{count}_{\ell, t, w, \text{add}}(id, id') = \text{DeletedSum}(\ell, id, id').
\]
2) Return the corresponding cnt value for id'.

Finds the hole and returns the cnt value right after id'.
Algorithm DeletedSum(ℓ, id, id')
1) \( \text{sum} := 0 \).
2) For each level \( ℓ' < ℓ \):
   - Find the region \([([ℓ, id_x], ([ℓ, id_y])); that falls within the range \([([ℓ, id], ([ℓ, id'])]\) (through binary search), and compute
     \( r := \text{count}_{ℓ', (id, id)}(id_x, id_y) \).
   - \( \text{sum} := \text{sum} + r \).
3) Return \( \text{sum} \).

Computes the hole
Improving Search

Documents containing keyword \( w \) added to \( L_4 \).

The range \([id, id']\) is a hole if:

\[
\text{count}_{L_4, L_4, w, \text{add}}(id, id') = \\
\text{count}_{L_3, L_4, w, \text{del}}(id_3, id_3') + \\
\text{count}_{L_2, L_4, w, \text{del}}(id_2, id_2') + \\
\text{count}_{L_1, L_4, w, \text{del}}(id_1, id_1') + \\
\text{count}_{L_0, L_4, w, \text{del}}(id_0, id_0')
\]

Removals of documents containing keyword \( w \) from target level \( L_4 \).

Remember
Improving Search

❖ Is $O(m\log^3 N)$ enough?
❖ We promised for $O(\min(\alpha + \log N, m\log^3 N))$
❖ How can we achieve this?

Remember Search algorithm from the Basic Structure. We will combine this algorithm with our new Search algorithm from Advanced Structure.
Improving Search - Hybrid Algorithm

1. Execute S1 until $O(\log^2 N)$ addition entries for keyword $w$ are encountered.

2. Through binary search each level $l$ find the total number $\alpha$ of addition entries referring to keyword $w$.

3. If $\alpha = w(\log^3 N)$, then execute S2; else S1.

If the total number of addition entries $\alpha = O(\log^2 N)$, the algorithm will terminate in the step 1 in $O(\alpha + \log N)$.

Otherwise, the algorithm will compute the exact number $\alpha$ in step 2, which takes $O(\log^2 N)$.

Step 3 Computes the min.
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.
Nearly each update needs log N documents transferred
## Database Size

<table>
<thead>
<tr>
<th>N</th>
<th>DB Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 million</td>
<td>20GB</td>
</tr>
<tr>
<td>100 million</td>
<td>42GB</td>
</tr>
<tr>
<td>200 million</td>
<td>82GB</td>
</tr>
<tr>
<td>400 million</td>
<td>130GB</td>
</tr>
<tr>
<td>800 million</td>
<td>202GB</td>
</tr>
</tbody>
</table>
The only difference between the basic algorithm and the advanced algorithm is efficiency.
Conclusion

❖ Still it is not practical enough
❖ If we add verifiability, keyword search takes microseconds
❖ Plenty room for improvement
Thank You

Questions?
The difference between the SimpleRebuild and Rebuild functions is SimpleRebuild downloads the entire level from the server, and locally computes the result and then uploads it back.

For the rebuild protocol we assume client has less storage.