Lectures 6+7: Zero-Leakage Solutions

Contents

1 Overview

In the first lecture we covered—at a high level—several ways to search on encrypted data. We also mentioned that for all solutions, there is a tradeoff between efficiency, security and expressiveness. Here we will cover one extreme of the solution space: the *zero-leakage* or *leakage-free* solutions. Note that by zero-leakage we do not mean that nothing is leaked. Rather, what we mean is that what is leaked can be derived efficiently from the security parameter.

We can build zero-leakage solutions from a primitive called oblivious RAM (ORAM). There are several ways to design ORAMs. The first is based fully-homomorphic encryption (FHE) and the second is based only on symmetric-key encryption (SKE). SKE-based ORAMs have received a lot of attention and there are many different constructions that achieve different levels of efficiency. In these notes we will discuss the FHE-based ORAM construction and the first and simplest SKE-based ORAM called the square-root solution. Then we will see how to use ORAM for encrypted search.

2 Oblivious RAM

An ORAM scheme structures an array in such a way that it can be *obliviously* accessed; that is, its items can be accessed (i.e., read and written) without revealing which items are accessed.

Definition 2.1 (Oblivious RAM). *An ORAM scheme* $\Omega = (\mathsf{Setup}, \mathsf{Read}, \mathsf{Write})$ *consists of three polynomial-time algorithms such that:*

- $\mathsf{ORAM} \leftarrow \mathsf{Setup}(1^k, \mathsf{RAM})$*: is a probabilistic algorithm that takes as input a security parameter* 1^k *and an array* RAM *of* N *items and it outputs a secret key* K *and an oblivious RAM* ORAM.
- $(\mathsf{RAM}[i], \perp) \leftarrow \mathsf{Read}\big((K, i), \mathsf{ORAM}\big)$*: is a two-party protocol executed between a client and a server. The client runs the protocol with a secret key* K *and an index* i *as input while the server runs the protocol with an oblivious RAM* ORAM *as input. At the end of the protocol, the client receives* $\mathsf{RAM}[i]$ *and the server receives* $\perp$.
- $(\perp, \mathsf{ORAM}') \leftarrow \mathsf{Write}\big((K, i, v), \mathsf{ORAM}\big)$*: is a two-party protocol executed between a client and a server that works as follows. The client runs the protocol with a key* K*, an index* i *and a value* v *as input and the server runs the protocol with an oblivious RAM* ORAM *as input. At the end of the protocol, the client receives* $\perp$ *and the server receives an updated oblivious RAM.*

Intuitively, an ORAM scheme is oblivious if its accesse pattern does not reveal any information about which item is accessed. This is defined as follows.

Definition 2.2 (Obliviousness). *Let* $\mathsf{ORAM} = (\mathsf{Setup}, \mathsf{Read}, \mathsf{Write})$ *be an ORAM scheme and consider the following randomized experiment against a stateful* PPT *adversary* $\mathcal{A}$:

$\mathbf{Obl}_{\mathcal{A}}(k)$:

1. $\mathcal{A}$ *outputs two sequences of* t *operations*

$$\mathbf{op}_0 = (\mathsf{op}_{0,1}, \ldots, \mathsf{op}_{0,t}) \quad \text{and} \quad \mathbf{op}_1 = (\mathsf{op}_{1,1}, \ldots, \mathsf{op}_{1,t})$$

   where $\mathsf{op}_{0,i}, \mathsf{op}_{1,i} \in \{\mathsf{Read}, \mathsf{Write}\}$;
2. *a bit* $b \overset{\$}{\leftarrow} \{0,1\}$ *is sampled;*
3. *the operations of* $\mathbf{op}_b$ *are executed with* $\mathcal{A}$ *playing the role of the server;*
4. $\mathcal{A}$ *outputs a guess* b';
5. *if* $b' = b$ *the experiment returns* 1 *else it returns* 0.

We say that ORAM *is oblivious if for all* PPT *adversaries* $\mathcal{A}$,

$$\Pr\left[\,\mathbf{Obl}_{\mathcal{A}}(k) = 1\,\right] \leq \frac{1}{2} + \mathsf{negl}(k).$$

3 Oblivious RAM via FHE

The simplest way to design an ORAM is to use FHE. An FHE scheme is an encryption scheme that supports computations on encrypted data. To enable this FHE schemes have, in addition to the standard Gen, Enc and Dec algorithms, an evaluation algorithm Eval that evaluates functions on ciphertexts.

Though FHE schemes support *any* computation on encrypted data, the concrete constructions achieve this by supporting only two homomorphic operations: addition and multiplication. That is, given ciphertexts $\text{ct}_1 = \mathsf{Enc}_K(m_1)$ and $\text{ct}_2 = \mathsf{Enc}_K(m_2)$, one can compute $\text{ct}_+ = \mathsf{Enc}_K(m_1 + m_2)$ and $\text{ct}_\times = \mathsf{Enc}_K(m_1 \times m_2)$. It turns out, however, that any function that can be evaluated in polynomial time can be represented as a polynomial-size *arithmetic circuit* which is a circuit composed of only addition and multiplication gates. So, given any polynomial-time computable function f, we can represent it as a polynomial-size arithmetic circuit C_f, and we can evaluate it homomorphically by executing each of its gates using the homomorphic addition and multiplication operations supported by the scheme. We now define the syntax of FHE schemes.

Definition 3.1 (Fully-homomorphic encryption)**.** *A symmetric-key fully-homomorphic encryption scheme* $\mathsf{FHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *consists of four polynomial-time algorithms that works as follows:*

- $K \leftarrow \mathsf{Gen}(1^k)$*: is a probabilistic algorithm that takes as input a security parameter* 1^k *and outputs a secret key* K*.*
- $\text{ct} \leftarrow \mathsf{Enc}(K, m)$*: is a probabilistic algorithm that takes as input a secret key* K *and a message* $m \in \mathbb{M}_k$ *and outputs a ciphertext* ct*.*
- $\text{ct}' \leftarrow \mathsf{Eval}(C, \text{ct}_1, \ldots, c_t)$*: is a deterministic algorithm that takes as input an arithmetic circuit* Cf *and a sequence of ciphertexts* $\text{ct}_1, \ldots, \text{ct}_t$ *and outputs a ciphertext* ct'*.*
- $m := \mathsf{Dec}(K, \text{ct})$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* ct *and outputs a message* m*.*

We say that FHE *is correct if it is a correct symmetric encryption scheme and if for all* $k \in \mathbb{N}$*, for all* K *output by* $\mathsf{Gen}(1^k)$*, for all* $t = \mathsf{poly}(k)$*, for all* $m_1, \ldots, m_t \in \mathbb{M}_k$*, for all ciphertexts* $\text{ct}_1, \ldots, \text{ct}_t$ *such that* $\text{ct}_i \leftarrow \mathsf{Enc}(K, m_i)$*, for all polynomial-size circuits* C*,* $\mathsf{Dec}\Big(K, \mathsf{Eval}(f, \text{ct}_1, \ldots, \text{ct}_t)\Big) = f(m_1, \ldots, m_t)$*.*

Note on correctness. Even though the correctness definition above is valid, it allows trivial FHE constructions. For example, it would be satisfied by the following scheme which is clearly not interesting. Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a standard symmetric-key encryption scheme and let $\mathsf{FHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ be such that Gen and Enc are the same as in SKE. Now, define Eval to be the algorithm that just outputs $\text{ct}' = (C, \text{ct}_1, \ldots, \text{ct}_t)$ and Dec be the algorithm that decrypts $\text{ct}_1, \ldots, \text{ct}_t$, applies C to the plaintexts and returns the result. To rule out this kind of trivial construction (and some that are less trivial) we have to bound the size of the ciphertexts Eval is allowed to output. In these notes, however, we use FHE as a black-box so whether the correctness definition allows trivial constructions or not is not particularly important.

The construction. Let $\mathsf{FHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ be a CPA-secure fully-homomorphic encryption scheme. Then we can construct an ORAM as follows:

- $\mathsf{Setup}(1^k, \mathsf{RAM})$: generate a key for the FHE scheme by computing $K = \mathsf{FHE.Gen}(1^k)$ and encrypt RAM as $\mathrm{ct} = \mathsf{FHE.Enc}_K(\mathsf{RAM})$. Output (K, ORAM), where $\mathsf{ORAM} = \mathrm{ct}$.

- $\mathsf{Read}\Big((K, i), \mathsf{ORAM}\Big)$: the client encrypts its index i as $\mathrm{ct}_i = \mathsf{FHE.Enc}_K(i)$ and sends ct_i to the server. The server computes
$$\mathrm{ct}' = \mathsf{FHE.Eval}(C_f, \mathrm{ct}, \mathrm{ct}_i),$$
where f is a function that takes as input an array and an index i and returns the ith element of the array. The server returns ct' to the client who decrypts it to recover $\mathsf{RAM}[i]$.

- $\mathsf{Write}\Big((K, i, v), \mathsf{ORAM}\Big)$: the client encrypts its index i as $\mathrm{ct}_i = \mathsf{FHE.Enc}_K(i)$ and its value as $\mathrm{ct}_v = \mathsf{FHE.Enc}_K(v)$ and sends them both to the server. The server computes
$$\mathrm{ct}' = \mathsf{FHE.Eval}(C_g, \mathrm{ct}, \mathrm{ct}_i, \mathrm{ct}_v),$$
where g is a function that takes as input an array, an index i and a value v and returns the same array with the ith element updated to v.

It follows by the CPA-security of FHE that ORAM reveals no information about RAM besides its size to the server and that the Read and Write protocols reveal no information about the index and values to the server.

Efficiency. The obvious downside of this FHE-based ORAM is efficiency. Currently, homomorphic evaluation is still expensive but is improving very rapidly. Many breakthroughs have happened in the past few years to decrease the cost of homomorphically evaluating a circuit. For our purposes, however, these improvements are not enough because the Read and Write protocols require $O(N)$ work by the server. That is, to read even a single item from ORAM the server has to (homomorphically) "touch" every item. Note that this is inherent in how FHE works because it evaluates circuits.

That being said, it is important to realize that even though using FHE as a black-box for this problem might not be efficient it does not mean that it could be effectively used as a building block in a larger solution that is practical.

4 Oblivious RAM via Symmetric Encryption

Fortunately, we also know how to design ORAMs using standard encryption schemes and, in particular, using symmetric encryption. ORAM is a very active area of research and we now have many constructions, optimizations and even implementations. Here, we will describe the simplest SKE-based construction known as the Square-Root solution. Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme and let F be a pseudo-random function that maps $\log N$ bits to $2\log N$ bits.

4.1 Setup

To setup the ORAM, the client generates two secret keys K_1 and K_2 for SKE and F, respectively. It then augments each item in RAM by appending its address and a random tag to it. We will refer to the address embedded with the item as its *virtual* address. More precisely, it creates a new array RAM_2 such that for all $1 \leq i \leq N$,

$$\mathsf{RAM}_2[i] = \left\langle \mathsf{RAM}[i], i, \mathsf{tag}_i \right\rangle,$$

where $\langle \cdot, \cdot, \cdot \rangle$ denotes concatenation and $\mathsf{tag}_i = F_{K_2}(i)$. It then adds $\sqrt{N}$ *dummy* items to RAM_2, i.e., it creates a new array RAM_3 such that for all $1 \leq i \leq N$, $\mathsf{RAM}_3[i] = \mathsf{RAM}_2[i]$ and such that for all $N + 1 \leq i \leq N + \sqrt{N}$,

$$\mathsf{RAM}_3[i] = \left\langle 0, \infty_1, \mathsf{tag}_i \right\rangle,$$

where ∞_1 is some number larger than $N + 2\sqrt{N}$. It then sorts RAM_3 around according to the tags. Notice that the effect of this sorting will be to permute RAM_3 since the tags are (pseudo-)random. It then encrypts each item in RAM_3 using SKE. In other words, it generates a new array RAM_4 such that, for all $1 \leq i \leq N + \sqrt{N}$,

$$\mathsf{RAM}_4[i] = \mathsf{Enc}_{K_1}(\mathsf{RAM}_3[i]).$$

Finally, it appends $\sqrt{N}$ elements to RAM_4 each of which contains an SKE encryption of 0 under key K_1. Needless to say, all the ciphertexts generated in this process need to be of the same size so the items need to be padded appropriately. The result of this, i.e., the combination of RAM_4 and the encryptions of 0, is the oblivious array ORAM which is sent to the server. It will be useful for us to distinguish between the two parts of ORAM so we'll refer to the second part (i.e., the encryptions of 0) as the *cache*.

4.2 Read & Write

Now we will see how to read and write to ORAM *obliviously*, i.e., without the server knowing which array locations we are accessing. First we have to define two basic operations: Get and Put.

The Get operation takes an index $1 \leq i \leq N$ as input and works as follows:

1. the client requests from the server the item at virtual addres i in ORAM. To do this it first re-generates the item's tag $\mathsf{tag}_i = F_{K_2}(i)$. It then does an (interactive) binary search to find the item with virtual address i. In other words, it asks the server for the item stored at location $N/2$ (let 's assume N is even) decrypts it and compares its tag with tag_i. If tag_i is less than the tag of item $\mathsf{ORAM}[N/2]$, then it asks for the item at location $N/4$; else it asks for the item at location $3N/4$; and so on.

2. it decrypts the item with tag_i to recover $\mathsf{RAM}[i]$,

3. it then re-encrypts $\mathsf{RAM}[i]$ (using new randomness) and asks the server to store it back where it was found.

The Put operation takes an index $1 \leq i \leq N$ and a value v as inputs and works as follows:

1. the client requests from the server the item with tag_i (as above);
2. it then encrypts v and asks the server to store it back at the location where the previous item (i.e., the one with tag_i) was found.

Notice that from the server's point of view the two operations look the same. In other words, the server cannot tell whether the client is executing a Get or a Put operation since in either case all it sees is a binary search followed by a request to store a new ciphertext at the same location.

One-time obliviousness. Now suppose for a second that ORAM only consisted of RAM_4. If that were the case then ORAM would be one-time oblivious in the sense that we could use it to read or write only once by executing either a Get or a Put operation. Why is this the case? Remember that we randomly permuted and encrypted our array before sending it to the server. This means that asking the server for the item at location j reveals nothing about that item's real/virtual address i. Furthermore, the binary search we do when looking for the item with virtual address i depends only tag_i which is random and therefore reveals nothing about i. Of course, this only works once because if we want to access i again then we'll ask the server for the same location which immediately tells it something: namely, that we asked for the same thing twice.

Multi-time obliviousness. So how do we hide the fact that we're asking for the same thing twice? This is really the core difficulty in designing ORAMs and this is where the cache will come in. We start by initializing a counter $\mathsf{ctr} = 1$. To read location i we execute the following Read protocol:

1. We Get the entire cache. In other words, we execute $\mathsf{Get}(j)$ for all
$$N + \sqrt{N} + 1 \leq j \leq N + 2 \cdot \sqrt{N};$$
2. If any of the Get operations above result in the ith item (i.e., if we get an item with virtual address i) then we Get a dummy item by executing $\mathsf{Get}(N + \mathsf{ctr})$. Also, we set z to be the item we found in the cache and ℓ to be the cache location where we found it.
3. If none of the Get operations above resulted in the ith item, we execute a *modified* $\mathsf{Get}(i)$ and set z to be the result and $\ell = N + \sqrt{N} + \mathsf{ctr}$. The modified version of $\mathsf{Get}(i)$ works like a regular $\mathsf{Get}(i)$ operation, except that we update the item's virtual address to ∞_2, where $\infty_2 > \infty_1$. In other words, we store an encryption of $\langle \mathsf{RAM}[i], \infty_2, \mathsf{tag}_i \rangle$ back where we found it. This will be useful for us later when we'll need to re-structure ORAM.

4. We then process the entire cache again but slightly differently than before (we do this so that we can store the item in the cache for future accesses). In particular, for all $N + \sqrt{N} + 1 \leq j \leq N + 2 \cdot \sqrt{N}$,
   - if $j \neq \ell$ we execute a $\mathsf{Get}(j)$ operation
   - if $j = \ell$ we execute a $\mathsf{Put}(j, z)$.
5. We increase ctr by 1.

The first thing to notice is that this is correct in the sense that by executing this operation the client will indeed receive $\mathsf{RAM}[i]$. The more interesting question, however, is why is this oblivious and, in particular, why is this more than one-time oblivious? To see why this is oblivious it helps to think of things from the server's perspective and see why its view of the execution is independent of (i.e., not affected by) i.

First, no matter what i the client is looking for, it always Gets the entire cache so Step 1 reveals no information about i to the server. We then have two possible cases:

1. If the ith item is in the cache (at location ℓ), we Get a dummy item; and Put the ith item at location ℓ while we re-process the entire cache (in Step 4).
2. If the ith item is not in the cache, we Get the ith item and Put it in the next open location in the cache while we re-process the entire cache.

In either case, the server sees the same thing: a Get for an item at some location between 1 and $N + \sqrt{N}$ and a sequence of $\mathsf{Get}/\mathsf{Put}$ operations for all addresses in the cache, i.e., between $N + \sqrt{N}$ and $N + 2 \cdot \sqrt{N}$. Recall that the server cannot distinguish between Get and Put operations.

Write. The Write protocol is similar to the Read protocol. The only difference is that in Step 2, we set $z = v$ if the ith item is in the cache and in Step 3 we execute $\mathsf{Put}(i, v)$ and set $z = v$. Notice, however, that the Write protocol can introduce inconsistencies between the cache and RAM_4. More precisely, if the item has been accessed before (say, due to a Read operation), then a Write operation will update the cache but not the item in RAM_4. This is OK, however, as it will be taken care of in the re-structuring step, which we'll describe below.

4.3 Restructuring

So we can now read and write to the array without revealing which location we are accessing and we can do this more than once. The problem, however, is that we can do it at most $\sqrt{N}$ times because after that the cache is full. To go beyond $\sqrt{N}$, we need to *re-structure* the ORAM; that is, we have to re-encrypt and re-permute all the items in ORAM and reset our counter ctr to 1. If the client has enough space to store ORAM locally then it can just to download it, decrypt it to recover RAM, update RAM (in case there were any inconsistencies) and setup a new ORAM from scratch.

If, on the other hand, the client does not have enough local storage then the problem is much harder. Here we will assume the client only has $O(1)$ storage so it can store, e.g., only two items. Recall that in order to re-structure ORAM, the client needs to re-permute RAM_4 and re-encrypt everything obliviously while using only $O(1)$ space. Also, the client needs to do this in a way that updates the elements that are in an inconsistent state due to Write operations. The key to doing all this will be to figure out a way for the client to sort elements obliviously while using $O(1)$ space. Once we can obliviously sort, the rest will follow relatively easily.

Sorting networks. To do this, we use a *sorting network* which is a circuit composed of comparison gates. These gates take two inputs x and y and output the pair (x, y) if $x \leq_\lambda y$ and the pair (y, x) if $x \geq y$. Given a set of input values, the sorting network outputs the items in sorted order. Sorting networks have two interesting properties: (1) the comparisons they perform are independent of the input sequence; and (2) each gate in the network is a binary operation (i.e., takes only two inputs). Of course, there is an overhead to sorting obviously so Batcher's network requires $O(N \log^2 N)$ work as opposed to the traditional $O(N \log N)$ for sorting.

Oblivious sort. To obliviously sort a set of ciphertexts $(c_1, \ldots, c_{N+2\sqrt{N}})$ stored at the server, the client will start executing the sorting network and whenever it reaches a comparison gate between the ith and jth item, it will just request the ith and jth ciphertexts, decrypt them, compare them, and store them back re-encrypted in the appropriate order. Note that by the first property above, the client's access pattern reveals nothing to the server; and by the second property the client will never need to store more than two items at the same time.

Restructuring. Now that we can sort obliviously, let's see how to re-structure the ORAM. We will do it in two phases. In the first phase, we sort all the items in ORAM according to their virtual addresses. This is how we will get rid of inconsistencies. Remember that the items in RAM_3 are augmented to have the form $\langle \mathsf{RAM}[i], i, \mathsf{tag}_i \rangle$ for real items and $\langle 0, \infty_1, \mathsf{tag}_i \rangle$ for dummy items. It follows that all items in the cache have the first form since they are either copies or updates of real items put there during Read and Write operations.

So we just execute the sorting network and, for each comparison gate, retrieve the appropriate items, decrypt them, compare their virtual addresses and return them re-encrypted in the appropriate order. The result of this process is that ORAM will now have the following form:

1. the first N items will consist of the most recent versions of the real items, i.e., all the items with virtual addresses *other* than ∞_1 and ∞_2;
2. the next $\sqrt{N}$ items will consist of dummy items, i.e., all items with virtual address ∞_1.
3. the final $\sqrt{N}$ items will consist of the old/inconsistent versions of the real items, i.e., all items with virtual address ∞_2 (remember that in Step 3 of Read and Write we executed a modified $\mathsf{Get}(i)$ that updated the item's virtual address to ∞_2).

In the second phase, we randomly permute and re-encrypt the first $N + \sqrt{N}$ items of ORAM. We first choose a new key K_3 for F. We then access each item from location 1 to $N+\sqrt{N}$ and update their tags to $F_{K_3}(i)$. Once we have updated the tags, we sort all the items according to their tags. The result will be a new random permutation of items. Note that we don't technically have to do this in two passes; but it's easier to explain this way. At this point, we are done. ORAM is restructured and we access it again safely.

Efficiency. So what is the efficiency of the Square-Root solution? Setup is $O(N \log^2 N)$: $O(N)$ to construct the real, dummy and cache items and $O(N \log^2 N)$ to permute everything through sorting. Each access operation (i.e., Read or Write) is $O(\sqrt{N})$: $O(\sqrt{N})$ total get/put operations to get the cache twice and $O(\log N)$ for each get/put operation due to binary search. Restructuring is $O(N \log^2 N)$: $O(N \log^2 N)$ to sort by virtual address and $O(N \log^2 N)$ to sort by tag. Restructuring, however, only occurs once every $\sqrt{N}$ accesses. Because of this, we usually average the cost of re-structuring over the number read/write operations supported to give an amortized access cost. In our case, the amortized access cost is then

$$O\left(\sqrt{N} + \frac{N \log^2 N}{\sqrt{N}}\right)$$

which is $O(\sqrt{N} \cdot \log^2 N)$.

5 ORAM-Based Encrypted Search

Now that we can build an ORAM, we will see how to use it for encrypted search. There are at least two ways to do this.

A naive approach. The first is for the client to just dump all the n documents $\mathbf{D} = (D_1, \ldots, D_n)$ in an array RAM, compute $(K, \mathsf{ORAM}) \leftarrow \mathsf{Setup}(1^k, \mathsf{RAM})$ and send ORAM to the server. To search, the client simulates a sequential scan via the Read protocol; that is, it replaces every read operation of the scan with an execution of the Read protocol. To update the documents the client can similarly simulate an update algorithm using the Write protocol.

If we store $\mathbf{D}$ into an ORAM with block size B, this yields a solution with

$$O\left(\frac{|\mathbf{D}|}{B} \cdot \mathsf{T}\left(\frac{|\mathbf{D}|}{B}\right)\right)$$

block accesses for each search operation, where $\mathsf{T}(N)$ is the Read protocol's overhead (e.g., $\mathsf{T}(N) = O(\sqrt{N})$ for the Square-Root solution and $\mathsf{T}(N) = O(\log^3(N))$ for the Hierarchical solution). Clearly, such an approach would be completely impractical for all but very small datasets.

A better approach. A better idea is for the client to build two arrays RAM_1 and RAM_2. [1] In RAM_1 it stores a data structure that supports fast searches on the document collection (e.g., an inverted index) and in RAM_2 it stores the documents $\mathbf{D}$ themselves. It then builds and sends $\mathsf{ORAM}_1 \leftarrow \mathsf{Setup}(1^k, \mathsf{RAM}_1)$ and $\mathsf{ORAM}_2 \leftarrow \mathsf{Setup}(1^k, \mathsf{RAM}_2)$ to the server. To search, the client simulates a query to the data structure in ORAM_1 via the Read protocol (i.e., it replaces each read operation in the data structure's query algorithm with an execution of Read). From this, the client recovers the identifiers of the documents that contain the keyword and with this information it reads those documents from ORAM_2.

Care must be taken, however, in the choice of the underlying search structure. In particular, its search complexity must be input-independent otherwise the server can learn information from observing the *number* of memory accesses made to ORAM_1. So, for example, to use an optimal-time structure like an inverted index, the client has to pad the lists to be of the same size. A similar problem occurs for ORAM_2 where the client must always retrieve a fixed number of documents and a fixed number of blocks per document to hide the length of a document.

So let DB be the appropriately padded inverted index generated from $\mathbf{D}$. Assuming each node in the lists stores only a document ID, this approach yields a solution with

$$O\left(\frac{1}{B_1} \cdot \max_w \left\{\#\mathsf{DB}(w)\right\} \cdot \log n \cdot \mathsf{T}_1\left(\frac{|\mathsf{DB}|}{B_1}\right)\right)$$

block accesses to query ORAM_1 and

$$O\left(\frac{1}{B_2} \cdot \max_{w \in \mathsf{W}} \left\{\#\mathsf{DB}(w)\right\} \cdot \max_i \left\{|D_i|\right\} \cdot \mathsf{T}_2\left(\frac{|\mathbf{D}|}{B_2}\right)\right)$$

block accesses to retrieve the documents from ORAM_2, where $\mathsf{T}_1(N)$ and $\mathsf{T}_2(N)$ are the access overheads of ORAM_1 and ORAM_2 and B_1 and B_2 are their block sizes. Further assuming that $|\mathsf{DB}| \approx |\mathbf{D}|$ (which is reasonable in practice) and that the two ORAMs have the same overhead $T(N)$, the two-RAM solution is more efficient than the naive solution whenever

$$\max_{w \in \mathsf{W}} \left\{\#\mathsf{DB}(w)\right\} \cdot \left(\frac{\log n}{B_1} + \frac{\max_i |D_i|}{B_2}\right) \ll \frac{|\mathbf{D}|}{B},$$

which is very likely to occur in practice. Focusing then on the two-RAM solution, we note that by setting

$$B_1 = \max_{w \in \mathsf{W}} \left\{\#\mathsf{DB}(w)\right\} \cdot \log n,$$

the block access cost of a single query to ORAM_1 is roughly $O(\mathsf{T}_1(\#\mathsf{W}))$ since $|\mathsf{DB}|/B_1 \approx \#\mathsf{W}$.

[1] Of course, the following could be done using a single RAM, but splitting into two makes things easier to explain.

A concrete example. As a concrete example, we take the Enron dataset which includes 1.5×10^6 documents (emails and attachments). The full index of the dataset has 1.2×10^6 distinct keywords with a selectivity (i.e., the number of documents that contain the keyword) that ranges from 3 to 700×10^3. So for this dataset we have $n = 1.5 \times 10^6$, $\#\mathsf{W} = 1.2 \times 10^6$ and $\max_{w \in \mathsf{W}} \#\mathsf{DB}(w) = 700 \times 10^3$.

Assuming we use an ORAM with $B_1 = 4KB$ blocks, we can pack $32 \times 10^3 / \log(1.5 \times 10^6) = 1560$ document identifiers in a block. So each padded list will have block-length $(700 \times 10^3)/1560 = 448$. This results in ORAM_1 being of size $\#\mathsf{W} \cdot 448 = (1.2 \times 10^6) \cdot 448 \approx 537 \times 10^6$ blocks, or $2.15TB$. We note that an SSE index for the same dataset is on the order of $12GB$ according to [**?**]. Also, note that since the block-size of DB is about 537×10^6, the multiplicative factor in the search complexity of ORAM_1 is approximately 134 for ORAM solutions with $\log^2(N)/\log\log(N)$ overhead and 8122 for ORAM solutions with $\log^3(N)$ overhead.

Assuming a block size of $B_1 = 700 \times 10^3 \cdot \log(1.5 \times 10^6) \approx 10 \times 10^6$ bits or $1.25MB$ the number of block access per query to ORAM_1 will be on the order of $\mathsf{T}(1.2 \times 10^6)$ which is approximately 74 for ORAM solutions with $\log^2(N)/\log\log(N)$ overhead and 2742 for ORAM solutions with $\log^3(N)$ overhead.