Outsourced Bits

A research blog on cloud computing, cryptography, security, privacy, …

How to Search on Encrypted Data: Oblivious RAMs (Part 4)

This entry was posted on December 20, 2013, in Crypto Design, Encrypted Search, Privacy and tagged cloud storage, encrypted search, oblivious RAM, privacy, search on encrypted data. Bookmark the permalink. 21 Comments

(https://cloudcrypto.files.wordpress.com/2013/10/sse.jpg)

This is the fourth part of a series on searching on encrypted data. See parts 1 (http://outsourcedbits.org/2013/10/06/how-to-search-on-encrypted-data-part-1/), 2 (http://outsourcedbits.org/2013/10/14/how-to-search-on-encrypted-data-part-2/) and 3 (http://outsourcedbits.org/2013/10/30/how-to-search-on-encrypted-data-part-3/).

In the previous posts we covered two different ways to search on encrypted data. The first was based on property-preserving encryption (in particular, on deterministic encryption), achieved *sub-linear* search time but had weak security properties. The second was based on functional encryption, achieved *linear* search time but provided stronger security guarantees.

We'll now see another approach that achieves the strongest possible levels of security! But first, we need to discuss what we mean by security.

Security

So far, I have discussed the security of the encrypted search solutions informally—mostly providing intuition and describing possible attacks. This is partly because I'd like this blog to remain comprehensible to readers who are not cryptographers but also because formally defining the security properties of encrypted search is a bit messy.

So, which security properties should we expect from an encrypted search solution? What about the following:

1. the encrypted database EDB generated by the scheme should not leak any information about the database DB of the user;
2. the tokens t_w generated by the user should not leak any information about the underlying search term w to the server.

This sounds reasonable but there are several issues. First, this intuition is not precise enough to be meaningful. What I mean is that there are many details that impact security that are not taken into account in this high-level intuition (e.g., what does it mean not to leak, how are the search terms chosen exactly). This is why cryptographers are so pedantic about security definitions—the details really do matter.

Putting aside the issue of formality, another problem with this intuition is that it says nothing about the search results. More precisely, it does not specify whether it is appropriate or not for an encrypted search solution to reveal to the server which encrypted documents match the search term. We usually refer to this information as the client's *access pattern* and for concreteness you can think of it as the (matching) encrypted documents' identifiers or their locations in memory. All we really need as an identifier is some per-document unique string that is independent of the contents of the document and of the keywords associated with it.

So the question is: Is it appropriate to reveal the access pattern? There are two possible answers to this question. On one hand, we could argue that it is fine to reveal the access pattern since the whole point of using encrypted search is so that the server can return the encrypted documents that match the query. And if we expect the server to return those encrypted documents then it clearly has to know which ones to return (though it does not necessarily need to know the contents).

On the other hand, one could argue that, in theory, the access pattern reveals some information to the server. In fact, by observing enough search results the server could use some sophisticated statistical attack to infer something about the client's queries and data. Note that such attacks are not completely theoretical and in a future post we'll discuss work that tries to make them practical. Furthermore, the argument that the server needs to know which encrypted documents match the query in order to return the desired documents is not technically true. In fact, we know how to design cryptographic protocols that allow one party to send items to another without knowing which item it is sending (see, e.g., private information retrieval and oblivious transfer).

Similarly, we know how to design systems that allow us to read and write to memory without the memory device knowing which locations are being accessed. The latter are called *oblivious RAMs* (ORAM) and we could use them to search on encrypted data *without revealing the access pattern to the server*. The issue, of course, is that using ORAM will slow things down.

So really the answer to our question depends on what kind of tradeoff we are willing to make between efficiency and security. If efficiency is the priority, then revealing the access pattern might not be too much to give up in terms of security for certain applications. On the other hand, if we can tolerate some inefficiency, then it's always best to be conservative and not reveal anything if possible.

In the rest of this post we'll explore ORAMs, see how to construct one and how to use it to search on encrypted data.

Oblivious RAM

ORAM was first proposed in a paper by Goldreich and Ostrovsky [GO96 (http://www.cs.ucla.edu/~rafail/PUBLIC/09.pdf)] (the link is actually Ostrovsky's thesis which has the same content as the journal paper) on software protection. That work turned out to be really ahead of its time as several ideas explored in it turned out to be related to more modern topics like cloud storage.

An ORAM scheme $(\mathrm{Setup}, \mathrm{Read}, \mathrm{Write})$ consists of:

- A setup algorithm Setup that takes as input a security parameter 1^k and a memory (array) RAM of N items; it outputs a secret key K and an oblivious memory ORAM.
- A two-party protocol Read executed between a client and a server that works as follows. 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 memory ORAM as input. At the end of the protocol, the client receives $\mathrm{RAM}[i]$ while the server receives $\perp$, i.e., nothing. We'll write this sometimes as $\mathrm{Read}((K, i), \mathrm{ORAM}) = (\mathrm{RAM}[i], \perp)$.
- A two-party protocol Write 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 memory ORAM as input. At the end of the protocol, the client receives nothing (again denoted as $\perp$) and the server receives an updated oblivious memory ORAM' such that the i th location now holds the value v. We write this as $\mathrm{Write}((K, i, v), \mathrm{ORAM}) = (\perp, \mathrm{ORAM}')$.

Oblivious RAM via FHE

The simplest way to design an ORAM is to use fully-homomorphic encryption (FHE). For an overview of FHE see my previous posts here (http://outsourcedbits.org/2012/06/26/applying-fully-homomorphic-encryption-part-1/) and here (http://outsourcedbits.org/2012/09/29/applying-fully-homomorphic-encryption-part-2/).

Suppose we have an FHE scheme $\mathrm{FHE} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Eval}, \mathrm{Dec})$. Then we can easily construct an ORAM as follows [1]:

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

- $\text{Read}\big((K, i), \text{ORAM}\big)$: the client encrypts its index i as $c_i = \text{FHE.Enc}_K(i)$ and sends c_i to the server. The server computes

$$c' = \text{FHE.Eval}(f, \text{ORAM}, c_i),$$

  where f is a function that takes as input an array and an index i and returns the i th element of the array. The server returns c' to the client who decrypts it to recover $\text{RAM}[i]$.

- $\text{Write}\big((K, i, v), \text{ORAM}\big)$: the client encrypts its index i as $c_i = \text{FHE.Enc}_K(i)$ and its value as $c_v = \text{FHE.Enc}_K(v)$ and sends them both to the server. The server computes

$$c' = \text{FHE.Eval}(g, \text{ORAM}, c_i, c_v),$$

  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 i th element updated to v.

The security properties of FHE will guarantee that ORAM leaks no information about RAM to the server and that the Read and Write protocols reveal no information about the index and values either.

The obvious downside of this FHE-based ORAM is efficiency. Let's forget for a second that FHE is not practical yet and let's suppose we had a very fast FHE scheme. This ORAM would still be too slow simply because the homomorphic evaluation steps in the Read and Write protocols require $O(N)$ time, i.e., *time linear in the size of the memory*. Again, assuming we had a super-fast FHE scheme, this would only be usable for small memories.

Oblivious RAM via Symmetric Encryption

Fortunately, we also know how to design ORAMs using standard encryption schemes and, in particular, using symmetric encryption like AES. ORAM is a very active area of research and we now have many constructions, optimizations and even implementations (e.g., see Emil Stefanov's implementation

(http://www.emilstefanov.net/Research/ObliviousRam/)). Because research is moving so fast, however, there really isn't a good overview of the state-of-the-art. Since ORAMs are fairly complicated, I'll describe here the simplest (non-FHE-based) construction which is due to Goldreich and Ostrovsky [GO96 (http://www.cs.ucla.edu/~rafail/PUBLIC/09.pdf)]. This particular ORAM construction is known as the Square-Root solution and it requires just a symmetric encryption scheme $\mathrm{SKE} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$, and a pseudo-random function F that maps $\log N$ bits to $2 \log N$ bits.

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

$$\mathrm{RAM}_2[i] = \langle \mathrm{RAM}[i], i, \mathrm{tag}_i \rangle,$$

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

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

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 memory RAM_4 such that, for all $1 \leq i \leq N + \sqrt{N}$,

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

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 memory 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*.

Read & write. Now we'll see how to read and write to ORAM *obliviously*, i.e., without the server knowing which memory locations we're 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 $\mathrm{tag}_i = F_{K_2}(i)$. 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 $\mathrm{ORAM}[N/2]$, 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 $\mathrm{RAM}[i]$,
3. it then re-encrypts $\mathrm{RAM}[i]$ (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.

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 memory 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.

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 $\mathrm{count} = 1$. To read location i we execute the following Read protocol:

1. We Get the entire cache. In other words, we execute $\mathrm{Get}(j)$ 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 i th item (i.e., if we get an item with virtual address i) then we Get a dummy item by executing $\mathrm{Get}(N + \mathrm{count})$. 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 i th item, we execute a *modified* $\mathrm{Get}(i)$ and set z to be the result and $\ell = N + \sqrt{N} + \mathrm{count}$. The modified version of $\mathrm{Get}(i)$ works like a regular $\mathrm{Get}(i)$

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 $\langle \mathrm{RAM}[i], \infty_2, \mathrm{tag}_i \rangle$ 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 $\mathrm{Get}(j)$ operation
   - if $j = \ell$ we execute a $\mathrm{Put}(j, z)$.
5. We increase count 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 $\mathrm{RAM}[i]$.

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 $\mathrm{Get}/\mathrm{Put}$ 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. The Write protocol is similar to the Read protocol. The only difference is that in Step 2, we set $z = v$ if the i th item is in the cache and in Step 3 we execute $\mathrm{Put}(i, v)$ 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.

So we can now read and write to memory without revealing which location we're 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 so we have to stop.

Re-structuring. So what if we want to do more than $\sqrt{N}$ reads? In that case we need to *re-structure* our ORAM. By this, I mean that we have to re-encrypt and re-permute all the items in ORAM and reset our counter count to 1.

If the client has enough space to store ORAM locally then the easiest thing to do is just to download ORAM, decrypt it locally to recover RAM, update it (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 becomes harder. Here we'll 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.

To do this, Goldreich and Ostrovsky proposed to use a sorting network (http://en.wikipedia.org/wiki/Sorting_network) like Batcher's Bitonic network (http://en.wikipedia.org/wiki/Batcher's_sort). Think of a sorting network as a circuit composed of comparison gates. The gates take two inputs x and y and output the pair (x, y) if $x < y$ 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.

So 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 i th and j th item, it will just request the i th and j th 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.

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 $\langle \mathrm{RAM}[i], i, \mathrm{tag}_i \rangle$ for real items and $\langle 0, \infty_1, \mathrm{tag}_i \rangle$ 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 $\mathrm{Get}(i)$ 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've 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're done! ORAM is as good as new and we can start accessing 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 $O(\sqrt{N} \cdot \log^2 N)$.

ORAM-Based Encrypted Search

So now that we know how to build an ORAM, we'll see how to use it for encrypted search. There are two possible 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 , setup an ORAM $(K, \mathrm{ORAM}) = \mathrm{Setup}(1^k, \mathrm{RAM})$ and send ORAM to the server. To search, the client can just simulate a sequential search algorithm via the Read protocol; that is, replace every read operation of the search algorithm with an execution of the Read protocol. To update the documents the client can similarly simulate an update algorithm using the Write protocol.

This will obviously be slow. Let' s assume all the documents have bit-length d and that RAM has a block size of B bits. The document collection will then fit in (approximately) $N = n \cdot d \cdot B^{-1}$ blocks. The sequential scan algorithm is itself $O(N)$, but on top of that we'll have to execute an entire Read protocol for every address of memory read.

Remember that if we're using the Square-Root solution as our ORAM then the Read protocol requires $O(\sqrt{N} \cdot \log^2 N)$ *amortized* work. So in total, search would be $O(N^{3/2} \cdot \log^2 N)$ which would not scale. Now imagine for a second if we were using the FHE-based ORAM described above which requires $O(N)$ work for

each Read and Write. In this scenario, a single search would take $O(N^2)$ time!

A better approach. [2] A better idea is for the client to build two arrays RAM_1 and RAM_2 [3]. In RAM_1 it will store a data structure that supports fast searches on the document collection (e.g., an inverted index (http://en.wikipedia.org/wiki/Inverted_index)) and in RAM_2 it will store the documents $\mathbf{D}$ themselves. It then builds and sends $\mathrm{ORAM}_1 = \mathrm{Setup}(1^k, \mathrm{RAM}_1)$ and $\mathrm{ORAM}_2 = \mathrm{Setup}(1^k, \mathrm{RAM}_2)$ 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) [4]. From this, the client will recover the identifiers of the documents that contain the keyword and with this information it can just read those documents from ORAM_2.

Now suppose there are m documents that contain the keyword and that we're using an optimal-time data structure (i.e., a structure with a query algorithm that runs in $O(m)$ time like an inverted index). Also, assume that the data structure fits in N_1 blocks of B bits and that the data collection fits in $N_2 = n \cdot d/B$ blocks.

Again, if we were using the Square-Root solution for our ORAMs, then the first step would take $O(m \cdot \sqrt{N_1} \cdot \log^2 N_1)$ time and the second step will take

$$O\left(\frac{m \cdot d}{B} \cdot \sqrt{N_2} \cdot \log^2 N_2\right).$$

In practice, the size of a fast data structure for keyword search can be large. A very conservative estimate for an inverted index, for example, would be that it is roughly the size of the data collection [5]. Setting $N = N_1 = N_2$, the total search time would be

$$O\left((1 + d/B) \cdot m \cdot \sqrt{N} \cdot \log^2 N\right)$$

which is $O(m \cdot d \cdot B^{-1} \cdot \sqrt{N} \cdot \log^2 N)$ (since $d \gg B$) compared to the previous approach' s

$O(n \cdot d \cdot B^{-1} \cdot \sqrt{N} \cdot \log^2 N)$.

In cases where the search term appears in $m \ll n$ documents, this can be a substantial improvement.

Is This Practical?

If one were to only look at the asymptotics, one might conclude that the two-RAM solution described above might be reasonably efficient. After all it would take at least $O(m \cdot d \cdot B^{-1})$ time just to retrieve the matching files

from (unencrypted) memory so the two-RAM solution adds just a $\sqrt{N}$ multiplicative factor over the minimum retrieval time.

Also there are much more efficient ORAM constructions than the Square-Root solution. In fact, in their paper, Goldreich and Ostrovsky also proposed the Hierarchichal solution which achieves $O(\log^3 N)$ amortized access cost. Goodrich and Mitzenmacher [GM11 (http://arxiv.org/pdf/1007.1259v2.pdf)] gave a solution with $O(\log^2 N)$ amortized access cost and, recently, Kushilevitz, Lu and Ostrovsky [KLO12

(http://eprint.iacr.org/2011/327.pdf)] a solution with $O(\log^2 N / \log\log N)$ amortized cost (and there are even more recent papers that improve on this under certain conditions). There are also works that tradeoff client storage for access efficiency. For example, Williams, Sion and Carbunar [WSC08

(http://digitalpiglet.org/research/sion2008pir-ccs.pdf)] propose a solution with $O(\log N \cdot \log\log N)$ amortized

access cost and $O(\sqrt{N})$ client storage while Stefanov, Shi and Song [SSS12

(http://arxiv.org/pdf/1106.3652.pdf)] propose a solution with $O(\log N)$ amortized overhead for clients that have

$O(N)$ local storage, where the underlying constant is very small. There is also a line of work that tries to de-amortize ORAM in the sense that it splits the re-structuring operation so that it happens progressively over each access. This was first considered by Ostrovsky and Shoup in [OS97 (http://www.cs.ucla.edu/~rafail/PUBLIC/28.pdf)] and was further studied by Goodrich, Mitzenmacher, Ohrimenko, Tamassia [GMOT11 (http://arxiv.org/pdf/1107.5093.pdf)] and by Shi, Chan, Stefanov and Li [SSSL11 (http://eprint.iacr.org/2011/407.pdf)].

All in all this may not seem that bad and, intuitively, the two-RAM solution might actually be reasonably practical for small to moderate-scale data collections—especially considering all the recent improvements in efficiency that have been proposed. For large- or massive-scale collections, however, I'd be surprised [6].

Conclusions

In this post we went over the ORAM-based solution to encrypted search which provides the most secure solution to our problem since it hides everything—even the access pattern!

In the next post we'll cover an approach that tries to strike a balance between efficiency and security. In particular, this solution is as efficient as the deterministic-encryption-based solution while being only slightly less secure than the ORAM-based solution.

Notes

[1] I haven't seen this construction written down anywhere. It's fairly obvious, however, so I suspect it's been mentioned somewhere. If anyone knows of a reference, please let me know. Also, as Jon Katz points out in the comments, this approach requires the server to compute whereas traditional ORAM constructions do not. Some recent works have started to distinguish these variants of ORAM (see comments for more).

[2] Like the FHE-based ORAM, I have not seen this construction written down anywhere so if anyone knows of a reference, please let me know.

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

[4] Note that this will reveal to the server some information. If we're using an inverted index as the underlying search structure, it will reveal the number of documents that contain the keyword (see the comments for a brief discussion). The natural way to address this, of course, is to use an inverted index with lists that are padded to the maximum size.

[5] In practice, this would *not* be the case and, in addition, we could make use of index compression techniques.

[6] I won't attempt to draw exact lines between what's small-, moderate- and large-scale since I think that's a question best answered by experimental results.

21 thoughts on "How to Search on Encrypted Data: Oblivious RAMs (Part 4)"

1. Pingback: How to Search on Encrypted Data: Functional Encryption (Part 3) | Outsourced Bits

2. Pingback: How to Search on Encrypted Data: Deterministic Encryption (Part 2) | Outsourced Bits

3. Pingback: How to Search on Encrypted Data: Introduction (Part 1) | Outsourced Bits

4. **Aaron Iles says:**
   December 21, 2013 at 3:13 am
   Is there a type on the paragraph on Goldreich and Ostrovsky proposing a sorting network? Shouldn't the last sentence read 'Of course, there is an overhead to sorting obliviously …'?

   Reply

senykam says:
December 21, 2013 at 8:12 pm
thanks

Reply

5. **Aaron Iles says:**
December 21, 2013 at 3:19 am
How does an ORAM handle running out of space? If all N items were consumed (say for storing entries in an inverted index) could the ORAM being expanded for additional capacity?

Also, doesn't the ORAM impose the restriction that there is only one client? Otherwise all clients would be required to share symmetric keys after setup and restructuring? At least for each ORAM in use, you could obviously create a distinct ORAM dedicated to each client.

Reply

senykam says:
December 21, 2013 at 8:11 pm
I think most constructions are for a fixed capacity so one should over-provision when setting up the ORAM.

With respect to your second question, yes most constructions are for single clients though some papers do explore the multi-client setting. See, for example, the following paper by Goodrich, Mitzenmacher, Ohrimenko and Tamassia:
Click to access 1105.4125v1.pdf

Reply

6. **Prasanna says:**
March 27, 2014 at 8:27 am
an we combine FHE and Searchable algorithms(SE) for having efficient privacy preserving cryptosystem. if yes how. Plz let me know

If so which FHE or SHE or PHE scheme can be combined with searchable schemes to have a better cryptosystem, specially for cloud

Reply

senykam says:
May 26, 2014 at 6:20 pm
Not sure what you mean…adding FHE would make things considerably less efficient.

Reply

Prasanna says:
June 23, 2014 at 12:07 am
in searchable encryption schemes which scheme is better i.e
symmetric or asymmetric method

senykam says:
June 23, 2014 at 4:59 pm
It kind of depends on the context. Asymmetric can address some scenarios that symmetric cannot. On the other hand, it is less efficient and less secure. So it kind of depends on your particular application. Do you have anything specific in mind?

7. **jonkatz says:**

May 26, 2014 at 9:55 am
Nice post! A couple of comments:

In our recent PKC'14 paper, we made a distinction between ORAM schemes where the server does not do any computation (but only fetches blocks that the client requests) and "oblivious storage" protocols where the server may do computation. (I think the distinction is important, since for some applications of ORAM – e.g., a trusted CPU fetching data from untrusted memory — you need the server not to do computation.) If you buy this distinction, then your FHE-based construction is technically not an ORAM scheme (though it does suffice for what you want). In our paper, we also explore better variants of your FHE-based scheme that run in sublinear time.

Regarding your optimization [2], note that this leaks m to the server, whereas m is not leaked in the naive solution. So you are seeing a tradeoff between security and efficiency…

Reply

senykam says:
May 26, 2014 at 4:22 pm
Thanks Jon!

I think the distinction is indeed important to make. If I recall correctly, a few of the recent papers on non-interactive ORAM (e.g., by Lu & Ostrovsky, Williams & Sion etc.) make crucial use of computation at the server to achieve non-interaction so it's an important resource. The distinction also clearly matters in the context of secure co-processors. I think the distinction is less important in the context of cloud computing, however.

It is true that m is leaked. Olya Ohrimenko pointed this out to me before and I meant to update the post to make that clearer. In my mind, the way this approach would work is that the inverted index would be padded to the maximum m. This would leak only the maximum m and would increase the search complexity considerably.

Reply

8. **Sitaram Chamarty says:**
August 16, 2014 at 3:36 am
A few things come to mind, but please note I have not read any of the referenced papers, so maybe all that I say below has already been proven to be a crackpot idea or something!

I am also not a mathematician; these are just ideas that came to me when the square root thing got too complicated to grok easily 🙂

1. When doing the binary search, about half the time (all searches being equally likely) you'll stop after less than logN fetches from the server. (I'm using the word "fetch" to distinguish from "Get" in your post, which required O(logN) fetches).

But why stop there? For approximately a 30% penalty in number of fetches overall, you can — even after you found the tag you want — continue with the binary search, randomly choosing "less" or "greater" until the end, and discarding those fetches.

Do this for all Get operations, and the server doesn't know which of these logN items was the target in each case. The fake write can still happen — just write back one of those you picked, at random, not necessarily the target of the search.

2. Don’t always start the binary search from the item at N/2. Cache a few past hits (again, at random; not “last 4” or so but “some 4 among the last few searches). Use those to seed the binary search. Normally there are about N/2 search paths through the items (I don’t know the correct term for it, but for N=32 they are 16-8-4-2-1, 16-8-12-10-9, etc., in a binary search). By seeding the binary search from other, previously located values, this path becomes much more unpredictable.

3. A request for item at N/2 is a dead giveaway that a new Get is starting. Starting from some other point, as described above, helps to blur that boundary — at least on a reasonably busy system. The server doesn’t know if you’re continuing a binary search you already started or started a new one.

On lightly loaded systems, this won’t work. Timing can be used to infer that a new search is starting. Start a random search and discard items. Do this at iiregular intervals.

4. Randomly begin and end binary searches with a few fetches for random items. Even if long gaps between queries tip off that a new search has started, combined with not starting at N/2, this will prevent guessing which of the first few queries is the first fetch in the binary search.

I’m not a math whiz though, very far from it in fact, so I have no clue how to prove what level of protection, if any, these tricks give you. I’m just throwing out ideas that — intuitively — seem much faster than doing an O(sqrt(N)) number fetches for every Get, as the square root method requires.

(And yes, I did notice there are references to other papers, but have not looked at them).

Reply

senykam says:
October 7, 2014 at 9:20 am
Sitaram, I think some of the intuition behind your first proposal is related to how tree-based ORAMs work. This approach was first proposed here: http://www.iacr.org/archive/asiacrypt2011/70730197/70730197.pdf and has since been developed in a series of works (you can find more recent works by looking for papers that cite this paper on Google Scholar).

Reply

9. Pingback: How to Search on Encrypted Data: Searchable Symmetric Encryption (Part 5) | Outsourced Bits

10. Pingback: Applied Crypto Highlights: Restricted Oblivious RAMs and Hidden Volume Encryption | Outsourced Bits

11. **Toma says:**
February 27, 2016 at 3:56 am
Thank you for the article. I’d like to clarify just one thing. Does server have the secret key K? If we design ORAM via FHE then in order to iterate over ORAM (inside function f) we should know exact index i (not encrypted Ci). If server has K, it could encrypt numbers from 0 to N, compare them to Ci and thereby find i. I know it’s not secure but I don’t see how f could be constructed without K. Can you please clarify this for me?

Reply

12. **senykam says:**
March 2, 2016 at 10:25 am
Hi Toma. No the server does not have the key K. The iteration over the encrypted array using the function f is done *homomorphically* so this means that the server does the iteration over encrypted inputs! In other words, the server has as input the encrypted array, the encrypted index and then executes the function f over these ciphertexts. That is the magic that FHE gives us!

Reply

Toma says:

March 7, 2016 at 1:26 pm

Thank you for the explanation. But I couldn't come up with an idea of how to construct such magic function. As far as I know it's enough to have summations and multiplications to perform any operations. Honestly, I don't see a way how to reduce iteration over an encrypted array or access by encrypted index to additions and multiplications. Could you give me a hint here or recommend an article to read?

Reply

senykam says:

March 8, 2016 at 1:34 pm

The "simple" answer is that we know that any efficiently-computable function can be written down as a circuit of reasonable size. So, for theoretical purposes, you don't really need to worry about exactly how to design the circuit. If you wanted to actually run this computation you would write your function in some high-level language and then compile it into a boolean circuit and then into an arithmetic circuit.