Oblivious RAM: A Dissection and Experimental Evaluation

Authors: Zhao Chang, Dong Xie, Feifei Li
Presenter: Minying Meng
11/3/2016
Outline

• Introduction
• Problem definition
• Preliminaries
• Oblivious RAMs
• Experiments
• Other ORAMs
• Conclusion
Introduction

• A lot of “big” data
• Rise of cloud computing
• Outsourced data storage
• Security concerns behind outsourcing data to public clouds
Introduction

• Possible solutions

• Use “trusted” public cloud services

• Use private cloud infrastructure

• Use encryption
Introduction

• **Access patterns** of queries & operations leak data privacy and sensitive information

• When **similar sequence** of queries appears again, allow cloud to **predict** user actions

• Observing access patterns relates to privacy issues
Introduction

• Only hiding the content is not enough
  • E.g. In an online health application, when the user selects a health condition from a list, the server would send back an updated web page with information on that illness. By learning the size of the page or file access pattern for each health condition, they could determine which conditions a user had without seeing any data (even if encrypted)
Introduction

• Theoretical interests
  • Improve worst case theoretical bounds
    • Hide large constant factors
    • Hard to be implemented and used in practice

• Not been thoroughly compared with each other
• Never been experimentally tested against large data
• Not easy to be correctly implemented with efficiency and scalability
Problem definition

• Data are stored in **atomic units**, referred as **blocks**.
• Each block has an **unique ID**, Block + ID = **Item**
• **Capacity**: the total number of items that an OS instance needs to support.
• **Server**: a general key-value storage service supports:
  • $get(k), put(k, v)$: get/put a value to a specific key
  • $getRange(k_1, k_2, p)$: return the first $p$ items with keys in range $[k_1, k_2]$
  • $delRange(k_1, k_2)$: remove all items with keys within range $[k_1, k_2]$
• **Client**:
  • Holds **a small amount of private memory** (either $O(1)$ or $o(n)$)
  • User can pose $get(k)$ and $put(k, v)$ to the client to access her data
Preliminaries

• Oblivious sort
  • Sorts a set of items by accessing the items in a fixed, predefined order

• Oblivious hash
  • Hashes a set of items into an array of hash buckets
  • Relies on an oblivious sort method

• Oblivious random permutation
  • Selected at random with uniform probability from all possible permutations

• Oblivious storage
  • A practical ORAM implementation
Basic Square Root ORAM

get\(k\)/put\((k, v)\)

Server

\[ N + 2\sqrt{N} \text{ items} = N \text{ original items} + \sqrt{N} \text{ dummy items} + \sqrt{N} \text{ shelter locations} \]

Client

① \(get(h_r(k))\)

② append returned item to cache

③ \(get/update\) item in cache

Found in cache? Not found!
**Basic Square Root ORAM**

**Algorithm 1: Basic-SR ORAM**

1: Initialization: the first $N$ blocks in the cloud are encrypted data blocks with identifiers 1 through $N$ from client’s database; set the other $2\sqrt{N}$ blocks (i.e., blocks in dummy and shelter locations) to randomly generated blocks with identifiers $(N + 1)$ to $(N + 2\sqrt{N})$.

2: Initialize $p := 0$.

3: while true do

4:  \hspace{1em} $p := p + 1$.

5:  \hspace{1em} Perform an oblivious random permutation $\pi$ of the blocks in the first $(N + \sqrt{N})$ locations (data + dummy blocks), and relocate the block in location $i$ ($1 \leq i \leq N + \sqrt{N}$) to location $\pi(i)$.

6:  \hspace{1em} for $count := 1$ to $\sqrt{N}$ do

7:  \hspace{2em} $j := (p - 1)\sqrt{N} + count$;

8:  \hspace{2em} // Simulate the $j$th operation ($op_j$, $id_j$, $block_j$) of the client.

9:  \hspace{2em} Client retrieves the blocks in locations $(N + \sqrt{N} + 1)$ through $(N + 2\sqrt{N})$ (i.e., shelter locations) from the cloud and looks for the block with the identifier $id_j$.

10:  \hspace{2em} if the block with $id_j$ is found in a shelter location $x$ then

11:  \hspace{3em} Retrieve a dummy block in location $\pi(N + count)$.

12:  \hspace{3em} Re-encrypt this block and write it back to same location.

13:  \hspace{2em} else

14:  \hspace{3em} Set $x := -1$. Retrieve the block $id_j$ in location $\pi(id_j)$.

15:  \hspace{3em} Encrypt a randomly generated block and write it back in location $\pi(id_j)$.

16:  \hspace{2em} // Client asks the cloud to scan the shelter locations again.

17:  \hspace{1em} for $i := 1$ to $\sqrt{N}$ do

18:  \hspace{2em} Read the block at shelter location $(N + \sqrt{N} + i)$.

19:  \hspace{2em} if $(N + \sqrt{N} + i) = x$ then

20:  \hspace{3em} if $op_j = write$ then

21:  \hspace{4em} Write encrypted $block_j$ to location $(N + \sqrt{N} + i)$.

22:  \hspace{4em} else if $op_j = read$ then

23:  \hspace{5em} Write the re-encrypted block at location $(N + \sqrt{N} + i)$ back to the same location.

24:  \hspace{2em} else if $i = count$ and $x = -1$ then

25:  \hspace{3em} if $op_j = write$ then

26:  \hspace{4em} Write encrypted $block_j$ to location $(N + \sqrt{N} + i)$.

27:  \hspace{4em} else if $op_j = read$ then

28:  \hspace{5em} Write the re-encrypted block with $id_j$ (retrieved in Line 14) into location $(N + \sqrt{N} + i)$.

29:  \hspace{2em} else

30:  \hspace{3em} Write the re-encrypted block at location $(N + \sqrt{N} + i)$ back to location $(N + \sqrt{N} + i)$.

31:  \hspace{2em} Perform an oblivious sort on blocks in all $(N + 2\sqrt{N})$ locations, where the sorting order is based on the block identifiers.
Interleave Buffer Shuffle SR-ORAM

- Oblivious storage scheme
- Propose a new oblivious shuffle algorithm
Algorithm 2: IBS-SR ORAM

1: **Initialization**: choose a hash function $h$; the first $N$ blocks in the cloud storage are real data blocks attached with hash values $h(1)$ through $h(N)$; the other $\sqrt{N}$ blocks are dummy random blocks attached with the hash values $h(N+1)$ through $h(N+\sqrt{N})$.

2: Initialize $p := 0$.

3: Let $T := \sqrt{N} + \sqrt{N}$.

4: while true do

5:     $p := p + 1$.

6:     Choose a new hash function $h'$.

7:     for $i := 1$ to $T$ do

8:         Retrieve and delete the blocks with the hash values $h(((i-1)T + 1)$ through $h(iT)$.

9:         Update each block's hash value using $h'$.

10:    Perform a random permutation on these $T$ blocks using the new hash values.

11:    Insert these $T$ blocks back into the cloud storage.

12:    Choose a new hash function $h''$.

13:    for $i := 1$ to $T$ do

14:        Retrieve and delete the blocks with the hash values $h'(0 \cdot T + i), h'(1 \cdot T + i), \ldots, h'((T-1)T + i)$.

15:    Update each block's hash value using new hash function $h''$.

16:    Perform a random permutation on these $T$ blocks with the new hash values.

17:    Insert these $T$ blocks back into the cloud storage.

18:    for count := 1 to $\sqrt{N}$ do

19:        $j := (p-1) \cdot \sqrt{N} + \text{count}$;

20:        // Simulate the $j$th operation $(op_j, id_j, \text{block}_j)$ of the client.

21:        Look for the block with the identifier $id_j$ in the client buffer using the hash value $h''(id_j)$.

22:        if the block with identifier $id_j$ is found in client buffer then

23:            Retrieve a dummy block with hash value $h''(N + \text{count})$.

24:        else

25:            Retrieve the block with the hash value $h''(id_j)$ (i.e., with identifier $id_j$).

26:        Write all re-encrypted blocks in the client buffer to the cloud storage. Clear the client buffer.
Basic Hierarchical ORAM

• Ask cloud to organize the blocks into a hierarchical structure

• Server:
  • log N “levels” for N items. Level i contains 2^i buckets. Each bucket contains log N slots.

• Client:
  • scans & writes re-encrypted block back to level 1
  • data blocks in level 1 reshuffle into level 2 every 2 operations
  • Data blocks in level 2 reshuffle into level 3 every 4 operations
Basic Hierarchical ORAM

Algorithm 3: Basic-HR ORAM

1: **Initialization:** choose a different hash function for each level \( \ell \) 
   \( (2 \leq \ell \leq L) \); \( N \) data blocks are hashed into the corresponding hash 
   buckets in level \( L \); set all other blocks in any hash bucket from any 
   level to random dummy blocks.

2: Initialize \( j := 0 \).

3: while true do

4: \( j := j + 1 \);

5: // Simulate the \( j \)th operation \((op_j, id_j, block_j)\) of the client.

6: Scan both buckets in level 1 and look for the block \( id_j \). If 
   found, write a dummy block back; otherwise, write the 
   re-encrypted block back.

7: for \( \ell := 2 \) to \( L \) do

8: if the block \( id_j \) has already been found then

9: \hspace{1em} Scan a random bucket (retrieve all blocks in the bucket) 
   in level \( \ell \) and write the re-encrypted block back.

10: \hspace{1em} else

11: \hspace{2em} Scan all blocks in the hash bucket that block \( id_j \) might be 
   hashed into. If it is found in the bucket, write a dummy 
   block back; otherwise, write the re-encrypted block back.

12: if \( j \) is an odd number then

13: \hspace{1em} Write the re-encrypted block \( id_j \) into 1st bucket in level 1.

14: \hspace{1em} else

15: \hspace{2em} Write the re-encrypted block \( id_j \) into 2nd bucket in level 1.

16: Let \( d := \max \{1 \leq x < L | j \mod 2^x = 0\} \).

17: for \( \ell := 1 \) to \( d \) do

18: \hspace{1em} Pick a new hash function for level \( \ell + 1 \).

19: \hspace{2em} Shuffle data blocks, obliviously to cloud, in level \( \ell \) and level 
   \( \ell + 1 \) together into level \( \ell + 1 \) using the new hash function.
TP-ORAM

• Improved hierarchical ORAM by leveraging partitioning
• Subdivide the O-RAM into much smaller partitions
• the operations performed on the partitions can be handled much more efficiently.
• Each partition is a full functional ORAM scheme
TP-ORAM

- Position map to track which partition each item resides in
- Cache read/updated blocks in a random partition’s cache slot
- Evict items in cache slots periodically to its ORAM partition
Algorithm 4: TP-ORAM

1: **Initialization**: the client stash is empty; each of $N$ data blocks is assigned to an independently selected random partition.
2: Initialize $j := 0$. $s := 1$.
3: while true do
4:   $j := j + 1$;
5:   // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
6:   Let $r$ be a random integer in the range $[1, P]$.
7:   $p := position[id_j]$;
8:   $position[id_j] := r$;
9:   if block $id_j$ is found in $stash[p]$ then
10:      Read and delete block $id_j$ from $stash[p]$.
11:      Read a dummy block from partition $p$ in the cloud storage.
12:   else
13:      Read block $id_j$ from partition $p$ in the cloud storage.
14:      Add block $id_j$ into $stash[r]$.
15:      // Piggy-backed eviction
16:      if $stash[p]$ is empty then
17:         Write a dummy block to partition $p$ in the cloud storage.
18:      else
19:         Write a block from $stash[p]$ to partition $p$ in the cloud storage. Remove the block from $stash[p]$.
20:      // Sequential eviction
21:      if $|j - R| - 1$ if $j - 1$ if $R = 1$ then
22:         if $stash[s]$ is empty then
23:            Write a dummy block to partition $s$ in the cloud storage.
24:         else
25:            Write a block from $stash[s]$ to partition $s$ in the cloud storage. Remove the block from $stash[s]$.
26:         $s := (s \mod P) + 1$;
BASIC BINARY-TREE ORAM

• Cloud storage is treated as a binary tree
• Each data block is mapped to a leaf node
• Selected uniformly at random
Path-ORAM

• Optimized binary-tree ORAM
  • Organize data blocks on the server as a full binary tree (log \(N\) levels, \(N\) leaf nodes).
  • Each node in the tree is a bucket of \(Z\) items
  • Each item is assigned to a random leaf node of the tree.

• Difference: each bucket contains different blocks
Path-ORAM

• There is a **position map** to track which leaf node is assigned to a data item.

• **Retrieve the whole path** that may contain the item and push all items on the path in **client’s private stash**
**Algorithm 5: Path-ORAM**

1: **Initialization**: client stash $S$ is empty; buckets in cloud contain dummy random blocks (except locations with data blocks which are oblivious to cloud); for each data block $b$, its position map $position[b]$ is initialized to the leaf node index that it is mapped to.

2: Initialize $j := 0$.

3: while true do

4: \[ j := j + 1; \]

5: // Simulate the $j$th operation ($op_j, id_j, block_j$) of the client.

6: Let $r$ be a random integer in the range $[0, 2^L - 1]$.

7: \[ x := position[id_j]; \]

8: \[ position[id_j] := r; \]

9: Retrieve path $P(x)$; find block $id_j$ in a bucket from $P(x)$; insert all data blocks from $P(x)$ to stash $S$.

10: for $\ell := L$ to 0 do

11: Let $S'$ be a subset of $S$, where for each block with an identifier $s$ in $S'$, $P(x, \ell) = P(position[s], \ell)$.

12: if $|S'| > Z$ then

13: Delete some blocks from $S'$ and make $|S'| = Z$.

14: else

15: Append dummy random blocks to $S'$ and make $|S'| = Z$.

16: $S := S - S'$;

17: Writes the blocks in $S'$ back to the bucket location $P(x, \ell)$. 
Comparison of ORAMs

<table>
<thead>
<tr>
<th>ORAM Construction</th>
<th>Computation Overhead (^a)</th>
<th>Cloud Storage</th>
<th>Communication Round</th>
<th>Client Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amortized</td>
<td>Worst-Case</td>
<td>Amortized</td>
<td>Worst-Case</td>
</tr>
<tr>
<td>Basic-SR [13]</td>
<td>(O(n \log n))</td>
<td>(O(\sqrt{N \log N}))</td>
<td>(O(N \log N))</td>
<td>(O(N \log N))</td>
</tr>
<tr>
<td>Oblivious Sort</td>
<td>(O(\sqrt{N \log N}))</td>
<td>(O(N \log N))</td>
<td>(O(N))</td>
<td>(O(\sqrt{N \log N}))</td>
</tr>
<tr>
<td>Basic-HR [14]</td>
<td>(O(n \log n))</td>
<td>(O(\log^2 N))</td>
<td>(O(N \log^2 N))</td>
<td>(O(N \log^2 N))</td>
</tr>
<tr>
<td>Oblivious Sort</td>
<td>(O(\log^3 N))</td>
<td>(O(N \log N))</td>
<td>(O(N \log N))</td>
<td>(O(\log^3 N))</td>
</tr>
<tr>
<td>BB-ORAM [30]</td>
<td>(O(\log^2 N))</td>
<td>(O(\log^2 N))</td>
<td>(O(N \log N))</td>
<td>(O(\log^2 N))</td>
</tr>
<tr>
<td>Recursive</td>
<td>(O(\log^3 N))</td>
<td>(O(\log^3 N))</td>
<td>(O(N \log N))</td>
<td>(O(\log^3 N))</td>
</tr>
<tr>
<td>TP-ORAM [32]</td>
<td>(O(\log N))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Non-Recursive, Non-Concurrent</td>
<td>(O(\log N))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Non-Recursive, Concurrent</td>
<td>(O(\log N))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Recursive, Non-Concurrent</td>
<td>(O(\log^2 N / \log B))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(\log^2 N / \log B))</td>
</tr>
<tr>
<td>Recursive, Concurrent</td>
<td>(O(\log^2 N / \log B))</td>
<td>(O(\sqrt{N}))</td>
<td>(O(N))</td>
<td>(O(\log^2 N / \log B))</td>
</tr>
<tr>
<td>Recursion, Non-Concurrent</td>
<td>(O(N \log^2 N / \log B))</td>
<td>(O(N \log^2 N / \log B))</td>
<td>(O(1))</td>
<td>(O(N \log^2 N / \log B))</td>
</tr>
<tr>
<td>Recursion, Concurrent</td>
<td>(O(N \log^2 N / \log B))</td>
<td>(O(N \log^2 N / \log B))</td>
<td>(O(1))</td>
<td>(O(N \log^2 N / \log B))</td>
</tr>
<tr>
<td>Path-ORAM [33]</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

\(^a\)For each operation of the client, the number of blocks retrieved/stored in the cloud, the total communication overhead in bytes, the cost of encryption/decryption in the client, and the total running time in the client have the same \(\Omega(\log N)\) complexity. All of them are shown as the Computation Overhead.

\(^b\)In fact, for each of \(\log_2 N\) levels in Basic-HR [14], the client needs to store a hash function. Thus, it needs extra client storage to save \(O(\log N)\) hash functions. However, in the practical setting, this cost is much less than the size of a constant number of blocks.

\(^c\)The complexity of recursive TP-ORAM using Cuckoo Hashing [16] as the partition ORAM of theoretical interest.

\(^d\)The complexity of recursive TP-ORAM using the original partition ORAM as the blackbox partition ORAM in our implementation.

Table 2: Comparison of different ORAMs’ performance.
Experiments

• Two machines: client and server
  • Client: 6GB main memory
  • Server: 95GB main memory and 1TB hard disk

• Connected by 1Gbps Ethernet

• Storage engine: MongoDB on the server

• AES encryption + SHA2 hash provided by CryptoPP

• Implement different ORAM schemes in a unified testbed.
Cloud and Client Storage Costs

(a) cloud storage cost.  
(b) client storage cost.

Figure 1: Cloud and client storage costs.
Query Performance in the Cloud

Figure 2: Number of blocks accessed per operation in cloud.

(a) amortized cost.  
(b) worst-case cost.
Query Cost for the Client

(a) amortized cost.
(b) worst-case cost.

Figure 3: Client-side query time per operation.

(a) amortized cost.
(b) worst-case cost.

Figure 4: Cost of encryption/decryption per operation.
Communication and End-to-End Cost

**Figure 5: Communication overhead in bytes per operation.**

- (a) amortized cost.
- (b) worst-case cost.

**Figure 6: Number of communication rounds per operation.**

- (a) amortized cost.
- (b) worst-case cost.

**Figure 7: End-to-end running time per operation.**

- (a) amortized cost.
- (b) worst-case cost.
Recursive ORAMs

Figure 8: Recursive ORAMs vs. non-recursive ORAMs: amortized cost used for (c), (d), and (e).
Using ORAM vs. Not Using ORAM

Figure 9: Using ORAM vs. not using ORAM (point query).

Figure 10: Using ORAM vs. not using ORAM (range query).
Other ORAMs

• Cuckoo Hashing ORAM
• Balanced ORAM
• Secure Multi-Party Computation
Conclusion #1

• Made a **comprehensive survey** on different ORAM constructions and principles.

• **Implement** different ORAM schemes in a **unified testbed**, and optimize them with respect to efficiency, scalability, and communication cost.

• Perform **extensive experiments on large data** to compare the performance of various ORAM constructions.
Conclusion #2

• Harm the performance of the database
  • Destroys any locality of references
  • Adds operation overheads

• Only support read and write operations
  • Complex queries may lead to large overhead
Thank you!