

## Lab 4: Path ORAM

*Instructor: Elaine Shi**Due Date: Nov 6, 11.59pm*

In this lab, you are going to individually implement Path ORAM and simulate your implementation. We expect this lab to be more time consuming than the previous ones, and simulations will take several hours, so we are giving extra time and we encourage you to **start early**.

## 1 Path ORAM

Path ORAM is an extremely simple Oblivious RAM protocol with a small amount of client storage. The goal of Oblivious RAM (ORAM) is to completely hide the data access pattern (which blocks were read/written) from the cloud storage server. From the server's perspective, the data access patterns from two sequences of read/write operations with the same length must be indistinguishable.

The research by Islam et al. [1] has demonstrated that an inference attack can identify as much as 80% of the search queries to an encrypted email repository by observing the access patterns. Therefore, concealing the access pattern to the remote storage server is of critical to protect the data privacy.

Existing ORAM algorithms prior to tree-based ORAM [3] achieve the goal by performing sophisticated deamortised oblivious sorting and oblivious cuckoo hash table constructions. Path ORAM is an improved tree-based ORAM. In contrast with previous work, Path ORAM is extremely simple since each ORAM access can be expressed as simply fetching and storing a single path in a tree stored remotely on the server.

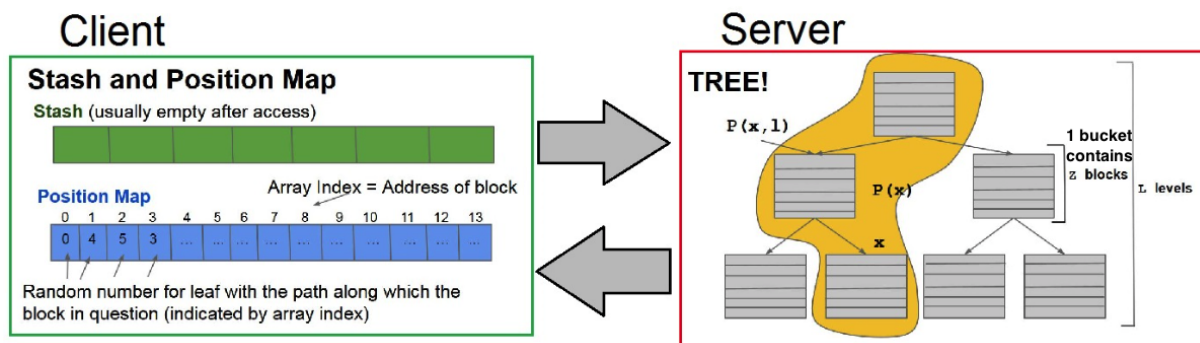


Figure 1: Visual representation of notations and data structures involved in the algorithm

|                           |  |
|---------------------------|--|
| $N$                       | Total # blocks outsourced to server  |
| $L$                       | Height of binary tree  |
| $B$                       | Block size (in bits)   |
| $Z$                       | Capacity of each bucket (in blocks)  |
| $\mathcal{P}(x)$          | path from leaf node $x$ to the root  |
| $\mathcal{P}(x, \ell)$    | the bucket at level $\ell$ along the path $\mathcal{P}(x)$   |
| $S$                       | client's local stash   |
| <b>position</b>           | client's local position map  |
| $x := \text{position}[a]$ | block $a$ is currently associated with leaf node $x$ , i.e., block $a$ resides somewhere along $\mathcal{P}(x)$ or in the stash. |

Figure 2: Notations

Figure 1 drawn by Zahur et al. [2] illustrates the data structures in Path ORAM that are maintained in the client side and the server side. We follow standard notations in Figure 2 defined by Path ORAM paper [4] to introduce the informal overview of its protocol. You should also read the original paper before starting with your labwork.

In general, data is divided into  $N$  blocks and encrypted then outsourced to a cloud server. Each block is represented as a simple data structure containing 3 fields: an ID  $a$  in the range  $[0, N)$ , a **leafID** that stores the identifier of the leaf of the path on which the block is placed, and a **data** field containing the actual data (in this homework, each block contains 24 bytes). Data are accessed in block units by either read or write operations.

On the server-side, data is stored in a tree, where each node is a bucket. Each bucket can contain up to  $Z$  real blocks. Although the tree does not have to be a binary tree, we will stick with a **complete binary tree** implementation in this labwork for simplicity. Let  $x \in \{0, 1, \dots, 2^L - 1\}$  denotes the  $x$ -th leaf node in the tree where  $L$  is **the length of a path** (levels are numbered in  $[0, L]$ ). Any leaf node  $x$  defines a unique path from leaf  $x$  to the root. We use  $\mathcal{P}(x)$  to denote a set of buckets along the path from leaf  $x$  to the root. Additionally,  $\mathcal{P}(x, \ell)$  denotes the bucket in  $\mathcal{P}(x)$  at  $\ell$ -th level in the tree.

The client stores two data structures, a stash and a position map:

**Stash.** During a read/write access, the stash stores blocks in  $\mathcal{P}(x)$  being retrieved by the client. After every access, the stash may still hold some blocks that are overflowed from the tree buckets on the server. It is usually empty (or contains a very small number of nodes) after a ORAM access completes.

**Position map.** Position map is an array that associates each block  $a$  to a leaf node (for example, block with ID  $a$  is associated with the leaf  $x := \text{position}[a]$ ). This means that block  $a$  resides in some bucket in the path  $\mathcal{P}(x)$  or in the stash. The position map is updated over time as blocks are accessed and remapped.

## 2 Implement Path ORAM Protocol

You are asked to implement two versions of the ORAM algorithm. For both versions, the client stash  $S$  is initially empty.

**The ORAM is initialized** by first setting the capacity of the remote storage to **contain enough buckets for a tree of height**  $L = \lceil \log_2(N) \rceil$ , where  $N$  is the number of blocks. For example, for an ORAM with  $2^{20}$  blocks, you will need a tree where  $L = 20$  (this means that the tree will typically have space to contain several times more blocks than the required size). Then, **all server buckets should be filled with**(random encryptions of) **dummy blocks (a dummy block is represented in this homework as a block whose ID and leaf id are -1 and whose data is set to 0)**, and the clients **position map is filled** with independently chosen random numbers between  $0$  and  $2^L - 1$  (please use the `RandomForORAMHW` class to sample those numbers).

**Reading and writing a block** to the ORAM is specified by a single algorithm called **Access**. Specifically, the client reads block  $a$  by performing  $\text{data} \leftarrow \text{Access}(\text{read}, a, \text{Null})$ . With respect to writing  $\text{data}^*$  to block  $a$ , the client performs  $\text{Access}(\text{write}, a, \text{data}^*)$ .

The **Access** protocol entails the following four high level steps in general:

- **Remap block:** Store the old position of block  $a$  in  $x$  before remapping this block to a new random position.
- **Read path:** Read the path  $\mathcal{P}(x)$  containing block  $a$  (if  $a$  is not stored along the path, then it must be in the stash).
- **Update block:** Update the data stored for block  $a$  if the access is a write operation.
- **Write path:** Write the blocks from the path back to the tree and possibly include some additional blocks from the stash if they can be placed into the path. (Blocks should be re-encrypted using fresh randomness so they appear new to the server). Buckets are greedily filled with blocks in the stash in the order of leaf to root, ensuring that blocks get pushed as deep down into the tree as possible. There are various methods to **evict blocks** (i.e. move blocks back) from the stash into the tree.

To make your life easier, in this homework blocks will **NOT be encrypted**, although a full-fledged implementation would have to do that. However, we ask you to **read and write back from the ORAM the buckets** even when no change to the data needs to be made. This is to simulate the fact that in all such cases, encryptions would need to be “refreshed”.

You are asked to implement the following two versions of Path ORAM in Java by completing the provided skeleton code. The two versions differ at the eviction method (i.e. how blocks are moved from the stash back into the tree).

During the implementation, instead of testing your code with a remote server, you can locally interact with an `UntrustedStorage` simulator provided by the skeleton code. We have also written a runnable example job in the `Job` class main method for your debugging.

**NOTE:** When you evict the blocks back into the tree, in both versions of the ORAM, you are requested to **respect this invariant** (which you will if you just implement the algorithm correctly from the given pseudocode) : each block  $a$  should only be placed in the tree on a leaf that is part of the path between the root and the leaf identified by `position[a]`. Submissions not compliant with this requirement will be particularly penalized.

## 2.1 Version 1: Read path eviction

In the ORAM with “read path eviction”, blocks are evicted from the stash on the same path that was read to retrieve the block. The pseudocode of the algorithm is shown in Algorithm 1. The skeleton code has set aforementioned variables like the bucket size  $Z$  as parameters. Therefore, you do **not** need to fix the value of variables when you are completing the skeleton code.

The first two lines **remap block  $a$** . Lines 3 to 5 describe the **read path** step. Lines 6 to 9 **update block  $a$** . The final lines 10 to 15 describe the **write path** step.

When performing eviction, it is important that blocks are placed in a position that is consistent with the position map (as that is only updated when a block is accessed). Therefore, block  $a'$  from the stash can be placed in the bucket at level  $\ell$  (on a specific path  $\mathcal{P}(x)$ ) only if the path  $\mathcal{P}(\text{position}[a'])$  to the leaf of block  $a'$  intersects path accessed  $\mathcal{P}(x)$  at level  $\ell$ . In other words, if  $\mathcal{P}(x, \ell) = \mathcal{P}(\text{position}[a'], \ell)$  at line 11. The next 2 lines of pseudocode means that if a bucket has less than  $Z$  real blocks, extra dummy blocks are stored to pad the bucket. If more than  $Z$  real blocks are to be stored, they are left on the client’s stash (and might be placed in other blocks in the following iterations).

## 2.2 Version 2: Deterministic, reverse lexicographical order eviction

In the second version of Path ORAM, blocks in the stash are evicted from a path different than the one they are read from. The path for eviction is chosen deterministically: a counter  $G$  is maintained (initially set to 0 and incremented after each access operation), and the leaf defining the path is the one with index given by “reversing the bits” of

---

**Algorithm 1** Access( $\text{op}, a, \text{data}^*$ ):

---

```
1:  $x \leftarrow \text{position}[a]$ 
2:  $\text{position}[a] \leftarrow \text{UniformDistribution}(0 \dots 2^L - 1)$ 

3: for  $\ell \in \{0, 1 \dots, L\}$  do
4:    $S \leftarrow S \cup \text{ReadBucket}(\mathcal{P}(x, \ell))$ 
5: end for

6:  $\text{data} \leftarrow \text{Read block } a \text{ from } S$ 
7: if  $\text{op} = \text{write}$  then
8:    $S \leftarrow (S - \{(a, \text{data})\}) \cup \{(a, \text{data}^*)\}$ 
9: end if

10: for  $\ell \in \{L, L - 1 \dots, 0\}$  do
11:    $S' \leftarrow \{(a', \text{data}') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(\text{position}[a'], \ell)\}$ 
12:    $S' \leftarrow \text{Select min}(|S'|, Z) \text{ blocks from } S'$ 
13:    $S \leftarrow S - S'$ 
14:    $\text{WriteBucket}(\mathcal{P}(x, \ell), S')$ 
15: end for

16: return  $\text{data}$ 
```

---

$G \bmod 2^L$  (hence the name reverse lexicographical order). For example, if  $L = 2$  and  $G = 1$ , its bit representation (with  $L$  bits) is 01, reversing it gives 10 which corresponds to 2 and therefore eviction is performed on the path  $\mathcal{P}(2)$  (Figure 3 shows the sequence of paths when  $L = 2$ ). This rule is chosen because two consecutive eviction paths have the least possible overlap.

The complete access protocol in this case is similar to the previous one and is described as Algorithm 2. Note that in this case, when reading the initial path on the tree, all the blocks that are read from the path (except for the one that is being accessed) are immediately written back to the same location (lines 4-11).

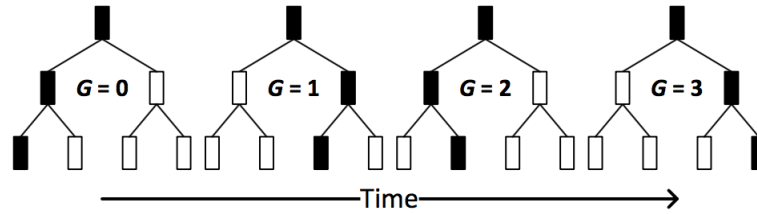


Figure 3: Reverse-lexicographic order of paths used by Algorithm 2. After path  $G = 3$  is evicted to, the order repeats.

---

**Algorithm 2** Access(*op*, *a*, *data*<sup>\*</sup>):

---

```
1: global G initialised to 0

2: x ← position[a]
3: position[a] ← UniformDistribution(0...2L - 1)

4: for  $\ell \in \{0, 1, \dots, L\}$  do
5:   bucket b ← ReadBucket( $\mathcal{P}(x, \ell)$ )
6:   if block a ∈ bucket b then
7:     b ← (b - {(a, data)}) ∪ {dummy block}
8:     S ← S ∪ {(a, data)}
9:   end if
10:  WriteBucket( $\mathcal{P}(x, \ell)$ , b)
11: end for

12: data ← Read block a from S
13: if op = write then
14:   S ← (S - {(a, data)}) ∪ {(a, data*)}
15: else
16: end if

17: g ← ReverseBits(G mod 2L)
18: G ← G + 1
19: for  $\ell \leftarrow \{0, 1, \dots, L\}$  do
20:   S ← S ∪ ReadBucket( $\mathcal{P}(g, \ell)$ )
21: end for

22: for  $\ell \leftarrow \{L, L - 1, \dots, 0\}$  do
23:   S' ← {(a', data') ∈ S :  $\mathcal{P}(g, \ell) = \mathcal{P}(\text{position}[a'], \ell)$ }
24:   S' ← Select min(|S'|, Z) blocks from S'
25:   S ← S - S'
26:   WriteBucket( $\mathcal{P}(g, \ell)$ , S')
27: end for

28: return data
```

---

## 2.3 Skeleton code package

You will download the zip file of skeleton code package from the CMS assignment page. Please do **not** change the package name and class names. Also, do **not** add any additional classes.

- Block.java

Block is the minimum unit of data in oblivious RAM. Each block should be 24-byte in the form of byte array. Each block should also store its index and its path's leaf ID. Remember that you need to use `System.arraycopy` to copy the data from one block to another. You should **not** modify this class.

- `Bucket.java`

Each node in the server storage tree is called a bucket. Each bucket can contain up to  $Z$  real blocks. You need to set the value of  $Z$  (max size of a bucket) before creating buckets. You must also implement all the incomplete methods for submission.

- `Job.java`

You can use this class to test your ORAM implementation for correctness but not for security. You can modify it for additional testing, but we will not take it into account. Instead, we will test with other jobs.

- `ORAMInterface.java`

This is a collection of abstract methods for Path ORAM. You should **not** modify this class.

- `ORAMWithDeterministicRLEViction.java`

You should complete the access protocol for Version 2 of Path ORAM in this class.

- `ORAMWithReadPathEviction.java`

You should complete the access protocol for Version 1 of Path ORAM in this class.

- `RandomForORAMHW.java` and `RandForORAMInterface.java`

You can call `getRandomLeaf` as the `UniformRandom` function in the pseudo-code. You should **not** modify these classes.

- `ServerStorageForHW.java` and `UntrustedStorageInterface.java`

`ServerStorage` is an instantiation of `UntrustedStorageInterface` with which ORAM needs to interact. You should **not** modify these classes.

### 3 Stash Size Analysis

The original Path ORAM algorithm does not state how big the client stash has to be in order for the algorithm to never stop. Indeed, it is possible that the client might have to even store *almost all* the data on the ORAM in the local stash (to see why, consider what would happen if the client is so unlucky that UniformDistribution in line 2 of the algorithm always returned the same leaf). However, if the client needs to reserve enough space to be able to locally store all the data from the ORAM, then the ORAM itself is not needed any more (it is indeed useful for devices with low memory that need to be able to work securely with large amounts of data).

However, since the algorithm is randomized, the stash size needed in practice will be much smaller than the worst case. This section will let you investigate empirically what is the appropriate size that needs to be allocated on the client for the ORAM.

#### 3.1 Task 3: Collecting the data

To begin with, you are asked to collect some data about how big the stash is during the normal usage of the ORAM. You will run your ORAM on as many access as you can (at least hundreds of millions), using a Job (similar to the one provided) that accesses (either for read or write) all the blocks of memory in sequence:  $\{1, 2, 3, 4, 5, \dots, N, 1, 2, 3, \dots, N, 1, 2, 3, \dots, N\}$ . This sequence was chosen because it is a worst case scenario in terms of the stash size. **After the first 3 000 000 (3 million) accesses (which are used to “warm up your ORAM” such that it enters steady distribution), please start recording the size of the stash after each access.**

At the end of the simulation, please write this data to a text file. The first line of the file should contain “-1,  $s$ ”, where  $s$  is the total number of accesses that you run the simulation for (excluding the first 3 millions as explained above). From the second line, each line should contain “ $i, s_i$ ”, where  $i$  is an integer representing the stash size, and  $s_i$  is the number of accesses after which the stash had size **greater than  $i$  (and not equal)** (starting with  $i = 0, 1, 2, \dots$  and up until when the maximum size of the stash you encountered is reached).

For example, if you run your ORAM for 3 000 003 accesses (remember that the first 3 000 000 are not counted because they are used to warm up the ORAM) and got stash size 1 after the 3 000 001<sup>st</sup> access, stash size 2 after the 3 000 002<sup>nd</sup> access, and again stash size 1 after the 3 000 003<sup>rd</sup> access, your file should look like:

```
-1,3
0,3
1,1
2,0
```

Note that the last  $s_i$  should always be 0 because this is the maximum stash size encountered, so the stash can never have a size greater than its maximum.



Using the previously collected data, you also need to **plot a figure** (and include it in the report) whose x-axis is  $R$  and y-axis is  $\log_2 \frac{1}{\delta(R)}$ , where  $\delta(R)$  is the **fraction of times** that the stash size exceeds  $R$ .

Please do all of the above for the following three configurations:

1.  $N = 2^{20}$ , bucket size  $Z = 4$  (for version 1: read path eviction)
2.  $N = 2^{20}$ , bucket size  $Z = 2$  (for version 2: deterministic order eviction)
3.  $N = 2^{20}$ , bucket size  $Z = 2$  (for version 1: read path eviction)

For each of them, submit a file for the cases A where  $s = 500\,000$  and B where  $s = 500\,000\,000$ . Please call your text files *simulationXY.txt*, where  $X = 1, 2, 3$  and  $Y = A, B$  (6 files in total). If you cannot do as many iterations as 500 000 000, then do as many as you can.

Using the previously collected data, you also need to **plot a figure** (and include it in the report) whose x-axis is  $R$  and y-axis is  $\log_2 \frac{1}{\delta(R)}$ , where  $\delta(R)$  is the **fraction of times** that the stash size exceeds  $R$  (6 plots in total).

### 3.2 Task 4: Bonus Question

Suppose we ask the question: what is a sufficiently confident estimate of the stash size corresponding to a  $2^{-20}$  failure probability? Obviously, simulating only  $2^{10}$  or  $2^{20}$  accesses won't be sufficient to obtain a "confident enough" estimate. How many accesses do we need in order to obtain a "confident enough" estimate of the stash size corresponding to a  $2^{-20}$  failure probability?

Please formally define your notion of confidence, and use mathematically rigorous reasoning to justify your answer.

## 4 Submission

- For the coding part of the assignment, you are required to submit `Bucket.java`, `ORAMWithDeterministicRLEviction.java` and `ORAMWithReadPathEviction.java` to CMS. Please make sure to fill in your NetID and name in all the source files that you submit (where requested).
- For task 3, you are required to submit the 6 `simulationXY.txt` files. Please make sure that they follow the prescribed format.

- You are required to submit a report in **PDF format**. The report should include the graphs/figures (12 in total) and the answers to the questions asked in the homework. Please **include your netid in all pages of the report**.
- You are also required to submit an additional zip file with all the code that you used to run the simulations for tasks 3 and 4, and to make the plots. The files inside the zip do not need to follow any naming convention, and you can use the language or tools that you are most comfortable with to generate the plots. If necessary, you can include a brief description of what you are submitting in the pdf report.
- Whenever an interface or abstract class is provided for you to interact with, your classes should **interact with the interface and not with the concrete implementation provided** (for example, your ORAM classes should declare the storage as `UntrustedStorage` and not as `ServerStorage`). For testing only (i.e. in the `Job` class that you can modify but that you are not required to submit, you are of course going to instantiate the concrete classes that provide the respective interfaces).
- You are encouraged to play around with the `Job` class to make sure your implementation is at least correct (i.e. reading and writing from the ORAM should give consistent results). Your ORAM should work with any parameter size and sequence of operations, not only the examples that we gave you.
- You are **NOT ALLOWED** to use non standard libraries without explicit permission of the instructor/TAs.

**Submission that is not compliant with the above instructions will be penalized.**

## References

- [1] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: RAMification, attack and mitigation.
- [2] Kate Highnam Samee Zahur, Matthew Irvine. Path ORAM, onion ORAM and garbled RAM, 2015. <https://s3.amazonaws.com/piazza-resources/i5d2rsalxrq1s4/i83x698kzdd48c/pathoramonion.pdf?AWSAccessKeyId=AKIAJKOQYKAYOBKKVTKQ&Expires=1444604527&Signature=YIMdKhGZHwtMd7xDopR8mdL11Qg%3D> [Online; accessed 11-October-2015].
- [3] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with  $O((\log N)^3)$  worst-case cost. In *Advances in Cryptology-ASIACRYPT 2011*, pages 197–214. Springer, 2011.

- [4] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.