CS 4830 – Introduction to Cryptography Mar 21, 2017

Lab 1: Path ORAM

Instructor: Elaine Shi ***Due Date: Apr 11, 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 homework, 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 Path ORAM achieve the goal by performing sophisticated deamortised oblivious sorting and oblivious cuckoo hash table constructions. 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.

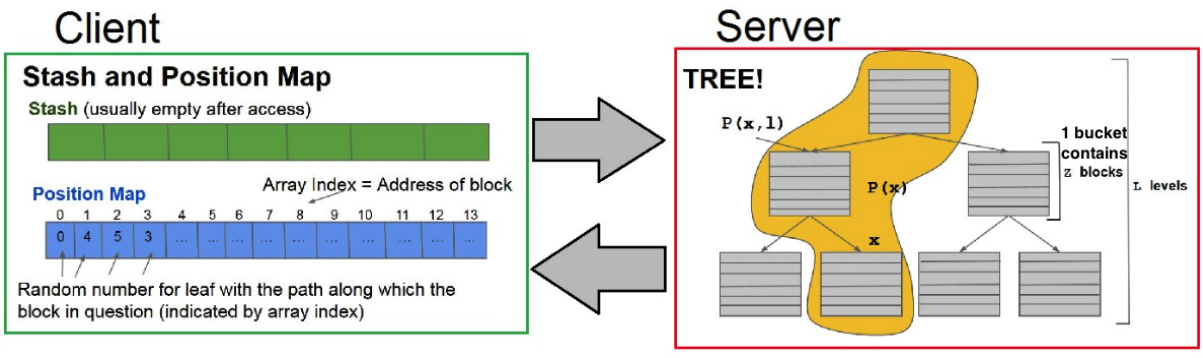

Figure 1: Visual representation of notations and data structures involved in the algorithm

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 ℓ along the path $\mathcal{P}(x)$
S	client's local stash
position	client's local position map
$x :=$ 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 [3] 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 contains B bits of data and is identified with an ID a in the range $[0, N)$. 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 $L = \lceil \log_2 N \rceil$ be the hieght of the tree (levels are numbered in $[0, L]$). Let $x \in \{0, 1, \ldots, 2^L - 1\}$ denotes the x-th leaf node in the tree. 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 ℓ-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 $x :=$ `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 filling all the server buckets with (random encryptions of) dummy blocks (a **dummy block** is represented in this homework as a block with ID -1 and data set to 0), and the clients position map is filled with independently chosen random numbers between 0 and $2^L - 1$ (please use the `RandomForOram` 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 `data` $\leftarrow$ `Access(read`, a, Null). With respect to writing `data`* to block a, the client performs `Access(write`, a, `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.

In this homework, blocks will **NOT be encrypted** (even though this makes the protocol insecure because the server can just read the memory, we did not talk about the required encryption schemes yet). 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.

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 ℓ (on a specific path $\mathcal{P}(x)$) only if the path $\mathcal{P}(\texttt{position}[a'])$ to the leaf of block a' intersects path accessed $\mathcal{P}(x)$ at level ℓ. In other words, if $\mathcal{P}(x, \ell) = \mathcal{P}(\texttt{position}[\texttt{a}'], \ell)$ 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: Determinstic, 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 mantained (initially set to 0 and incremented after each access operation), and the leaf defining the pat is the one with index given by "reversing the bits" of G mod 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).

Algorithm 1 Access(op, a, data*):

1: x $\leftarrow$ position[a]
2: position[a] $\leftarrow$ UniformDistribution$(0 \ldots 2^L - 1)$

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

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

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

16: **return** data

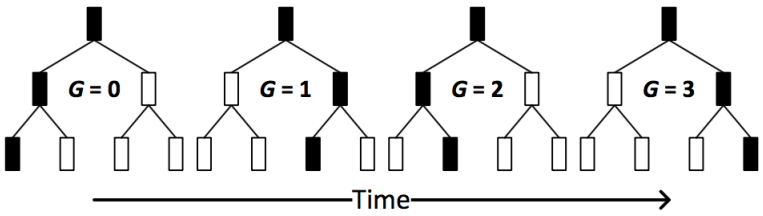

Figure 3: Reverse-lexicographic order of paths used by Algorithm 2. After path $G = 3$ is evicted to, the order repeats.

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 name.

- Block.java

  Block is the minimum unit of data in oblivious RAM. Each block should be 32-byte in the form of byte array. 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

Algorithm 2 Access(op, a, data*):

1: **global** G **initialised to** 0

2: x $\leftarrow$ position[a]
3: position[a] $\leftarrow$ UniformDistribution$(0 \dots 2^L - 1)$

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

12: data $\leftarrow$ Read block a from S
13: **if** op = write **then**
14: $S \leftarrow (S - \{(\text{a,data})\}) \cup \{(\text{a,data}^*)\}$
15: **else**
16: **end if**

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

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

28: **return** data

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, ... N, 1, 2, 3, ... N, 1, 2, 3, ... N}. This sequence was chosen because it is a worst case scenario in terms of the stash size. After the first 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** (starting with $i = 0$ 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 and got stash size 1 after the $3'000'001^{st}$ access, stash size 2 after the $3'000'002^{nd}$ access, and again stash size 1 after the $3'000'003^{rd}$ access, your file should look like:

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

Using the previously collected data, you also need to **submit a figure** (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)

Please call your text files *simulationX.txt*, where $X = 1, 2, 3$.

3.2 Task 4: Picking a good stash size

As you have seen in the previous task, the required stash size seems to be much smaller than the total number of blocks. In practical implementation, the stash of the clients

will be bounded (and much shorter than the worst case size) because devices have constrained memory. Therefore the Oram algorithm will fail (i.e. the stash will overflow and the algorithm will have to abort) whenever the stash size is bigger than that bound.

In this task, we are going to estimate, based on the measurements from the previous task, for a given (small) error probability ϵ, what is a good bound on the stash size so that (with 95% confidence based on the outcome of our simulation) the Oram will fails with probability smaller than ϵ.
In principle, this is a quite involved task, but for the purpose of this homework we will make some simplifying (but somewhat reasonable) assumptions: we assume that, given a bound on the stash size, and values for N and Z, the probability that after a specific access operation (during a given run of the protocol) the stash exceeds the bound is *independent* of the previous state of the Oram (and of the specific access pattern that we are using). Therefore, we can treat the event "During a run of the Oram algorithm (with parameters N,Z), the stash size exceeded bound B" as a Bernoulli distributed random variable, and perform a statistical test to asses whether the "failure" probability p is below ϵ.

In practice, you need to compute, for a given failure probability $\epsilon = 2^{-n}$, the value

$$F(n) = \min(\left\{ i \in \mathbb{N} \middle| \frac{s_i - s \cdot 2^{-n}}{\sqrt{s_i \cdot 2^{-n} \cdot (1 - 2^{-n})}} \leq z(0.95) \right\})$$

where $z(0.95) \approx 1.645$ is the 95^{th} percentile of the normal distribution, and s_i, s are taken from the experiments you run in the previous task.
Alternative. Another simpler way to estimate the stash size is to plot and extrapolate stash size i as follows: for each stash size i, compute $p_i = \frac{s_i}{s}$, and then plot i versus $-\log p_i$. Note that p_i is the (empirical) failure probability corresponding to ϵ in the previous method. (Same as defined in task 3, s is the total number of accesses, and s_i is the number of accesses such that stash size is greater than i.)

For each of the 3 configurations of task 3, please plot a figure for $F(n)$ when the simulation is run for 5000 and 5 hundred millions steps (excluding the initial warmup). Is there any difference in the plots depending on the number of iterations? Why do you think it is the case? (please **answer these questions in the report**).

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 3 `simulationX.txt` files. Please make sure that they follow the prescribed format.
- 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 file 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.

Submission that is not complaint 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://piazza-resources.s3.amazonaws.com/i5d2rsalxrq1s4/i83x698kzdd48c/pathoramonion.pdf?AWSAccessKeyId=AKIAIEDNRLJ4AZKBW6HA&Expires=1490155533&Signature=PmJWsd1xQq%2BbLW8cBWYnThXdttY%3D` [Online; accessed 21-March-2015].

[3] 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.