Spring 2020

An Efficient Three-Entity Oblivious RAM Protocol

Jikang Qu

An Efficient Three-Entity Oblivious RAM Protocol

Abstract:

Cloud storage services are becoming more widely used in recent decades. Lots of systems can protect stored information from being leaked by data encryption but it is difficult to protect data access pattern privacy with high efficiency compared to non-oblivious systems. Also, lots of Oblivious RAM (ORAM) systems developed by researchers are just proposed but not implemented in the real world due to high overheads in communication and storage. In this report, we implement a new Oblivious RAM (ORAM) system which dramatically decreases server storage capacity, lowers communication cost and reduces query delay. We also show that this ORAM system can protect client's data access patterns.

Background:

It is very well known that cloud services are widely used in industry because of customers' increasing demand for computing power. Cloud services provide lots of attractive features such as pay-as-you-go and economic efficiency. With the popularity of cloud services, clients also need to raise various privacy concerns.

It is obvious that data encryption is not enough to protect clients' privacy in accessing outsourced data. Data encryption can prevent storage from being leaked but the server can still get clients' access pattern. While a client tries to access some storage locations consecutively, the server will easily get information of which targets the client wants to access.

In the past decades, how to solve the problem of protecting data access patterns has attracted a large amount of interest from researchers and engineers. However, most of them were approaching this theoretically but not practically. It's still not easy to build an oblivious cloud storage system with performance comparable with a non-oblivious cloud storage system. To solve this issue, we need to promise that an oblivious cloud storage system has to protect clients' data access patterns, lower query delay by clients, lower storage costs by clients and especially the server, and lower communication cost between clients and the server.

Oblivious RAM (ORAM) algorithm was developed in a proposal by Goldreich [1] in 1987 and Ostrovsky [2] in 1990. It allows a client to hide its data access pattern to an untrusted server. It basically works by encrypting and shuffling data which the client has

accessed. After this proposal, lots of researchers and engineers have put in efforts to develop new ORAM schemes [3],[4],[5],[6],[7],[8],[9].

In this creative component, we propose and implement a new ORAM based cloud storage system called Three-Entity ORAM. We make use of the more and more popular adoption of hardware or software based trusted execution environment (TEE), such as Intel SGX enclave, to shift the workload from client to the TEE at the server side, and thus can significantly reduce the client-server communication cost as well as the storage-side storage cost.

Problem Definition:

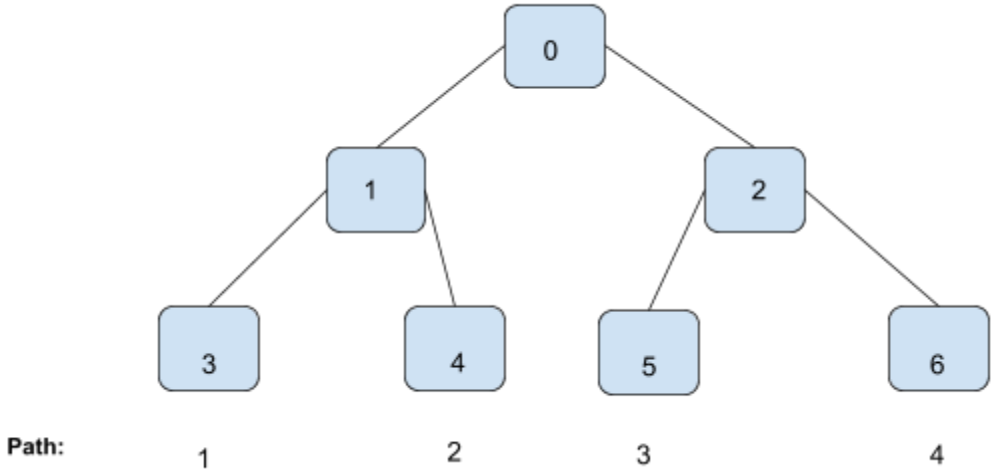
We consider a system composed of a client, a server, and a trusted entity (TEE) co-locating with the server. The client wants to store data at the server while protecting its privacy. Traditional encryption schemes only help hide data information and they can't hide clients' data access patterns. This means, blocks which are accessed and their order are revealed.

In our implementation, we assume that the client and the trusted entity are trusted, meaning all of their operations are honest and correct. We assume that the server is not trusted and it is curious about what blocks the client accessed. So, the goal of ORAM is to successfully and completely hide data access patterns from the server. Particularly, each block has to be in the same length which means the server can't distinguish blocks by its length.

We assume that the client fetches/stores data in atomic units. We call them blocks, and each includes an array of bytes. Also, the structure of our system is not that complicated even though it has three entities even unlike other normal 'Client-Server' ORAM. For security, we aim to hide: (1) which data is being accessed; (2) when data was last accessed; (3) whether the same data is being accessed or not; (4) whether the operation is read or write; and (5) access pattern. Note that we assume every data block is encrypted by the client and thus its content is not leaked.

Three-Entity ORAM System

Our System is composed of three entities: Client, Server and Trusted Entity. The figure below provides a brief example of server storage. There are seven buckets totally forming a binary tree. The tree has four paths, each having three buckets. For each bucket, there is an array of blocks. Each block owns data with the type of byte array that can be transferred into a string. Note that the example shows a binary tree, but other tree structural storage (m-ary tree) also works in this system.



For each bucket, it includes a fixed-size array of blocks. For all blocks, their lengths are also the same, which helps to prevent the server from tracing blocks based on their lengths. Also, initially, each bucket is fully filled with encrypted real and dummy blocks given by the client.

On the client side, the client has to store information about tree storage structure. It doesn't store information about the data of the block actually because there is not enough space for the client to store such a lot of data. This is also why cloud computing is very popular because lots of small companies rent cloud services from large companies which provide cloud with large storage (e.g. AWS, Azure, GCP). In our design, the client has to store two kinds of information based on server storage. First, the client has to store information about which buckets each path owns. For example, the client needs to know bucket {0,1,3} belongs to path 1. Second, the client has to store information about the location of each block. For example, the client has to know the block which ID is 0 located at bucket 0.

Particularly, unlike Path ORAM [3], the client does not need to query all blocks in one path on the server storage. In our implementation, the client only needs to query much fewer blocks on each bucket corresponding to its path. For example, if the target block is in path 1, then the client might just select 2 blocks from bucket 0, bucket 1 and bucket 3 for each. Assume there are 20 blocks in each bucket initially, Path ORAM [3] requires the client to query 60 blocks per query, but in our design, it only needs to select roughly 6 blocks per query.

Also, another improvement compared to Path ORAM [3] is, the client doesn't need to access all queried blocks and this makes no communication cost between the client and the server, which helps protect clients' data access patterns. The client once selects multiple blocks. It sends information to the trusted entity which includes block IDs (non-target and target), path ID, and operations(write or read). The trusted entity once gets the information from the client, it will query selected blocks on the server, and

then only sends the target block back to the client. There is a rule for the client to query and determine which blocks that the trusted entity will need to 'pretend' to access.

For each bucket Ni on the path which includes the target block, we need to check every block in Ni with three statuses: denoted as 0,1,2. If status is 0 means this block has never been accessed, which means the trusted entity never accessed this block and the client never selected it. If status is 1 means this block has been queried as a target before. If status is 2 means this block has been queried before but never a target. Δ here denotes each set in Ni and δ denotes the size of each set. The rule of query is shown in the figure below.

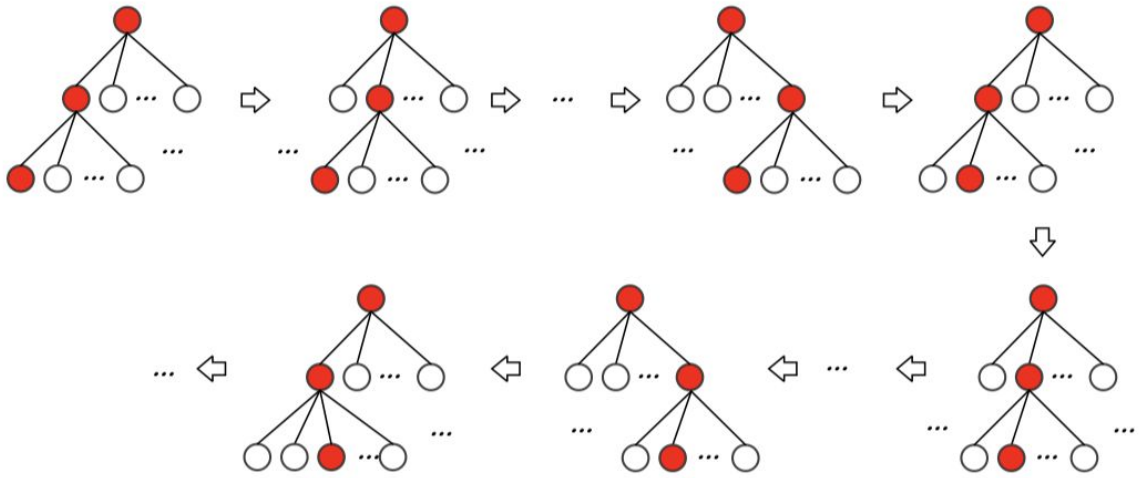
Algorithm 1 Rules for Selecting Blocks from $\mathcal{N}_i'$ to Access (Output: Δ - a set of blocks selected to access)

$\Delta \leftarrow \emptyset$
if $\mathcal{N}_i'$ contains query target $\vec{D}_t$ **then**
    add $\vec{D}_t$ to Δ
    $\forall \vec{D} \in \Delta_{i,1}$, add $\vec{D}$ to Δ with probability $\frac{1}{\delta_{i,0}}$
    if $\vec{D}_t$ belongs to $\Delta_{i,0}$ **then**
        $\forall \vec{D} \in \Delta_{i,2}$, add $\vec{D}$ to Δ with probability $\frac{\delta_{i,2}}{\delta_{i,0}^2}$
    else //i.e., $\vec{D}_t$ belongs to $\Delta_{i,2}$
        randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to Δ
else
    randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to Δ
    $\forall \vec{D} \in \Delta_{i,1} \cup \Delta_{i,2}$, add $\vec{D}$ to Δ with probability $\frac{1}{\delta_{i,0}}$

At the beginning, the client, based on the rule of query, selects multiple blocks, denoted as an array of block ID, {i1,i2,i3…,target ID}. Then, the client sends this information to the trusted entity, also, with the path ID and operation(read or write). Once the trusted entity hears the request from the client, it will query required blocks from the server and send the data of the target block to the client and wait for the client's operation to be done. If it's read operation, when the client finishes operation and sends the target block back to the trusted entity, the trusted entity does not have to do anything but only store it in local storage of itself. If it's write operation, when the

client finishes operation and sends the target block back to the trusted entity, the trusted entity has to encrypt the target block and store it in local storage of itself because the data is changed actually after the client's writing. We assume the client and the trusted entity share the key of encryption algorithm so that we can ensure the correctness of this system. In each process of query, after the trusted entity queries every target block, it has to make it into a dummy block at its original position in server storage because when the trusted entity puts every target block back the server, it needs to set each target a new random position.

After N queries by the client, the trusted entity will store N blocks which are targets. Then, the trusted entity has to evict all blocks which are in one path back to the server. For every target stored on the trusted entity side, the trusted entity randomly generates a new path for it, and then from the root to the leaf on the path, finds an empty slot and puts the target on it. The eviction path is reverse lexicographic order and it's shown in the figure below.



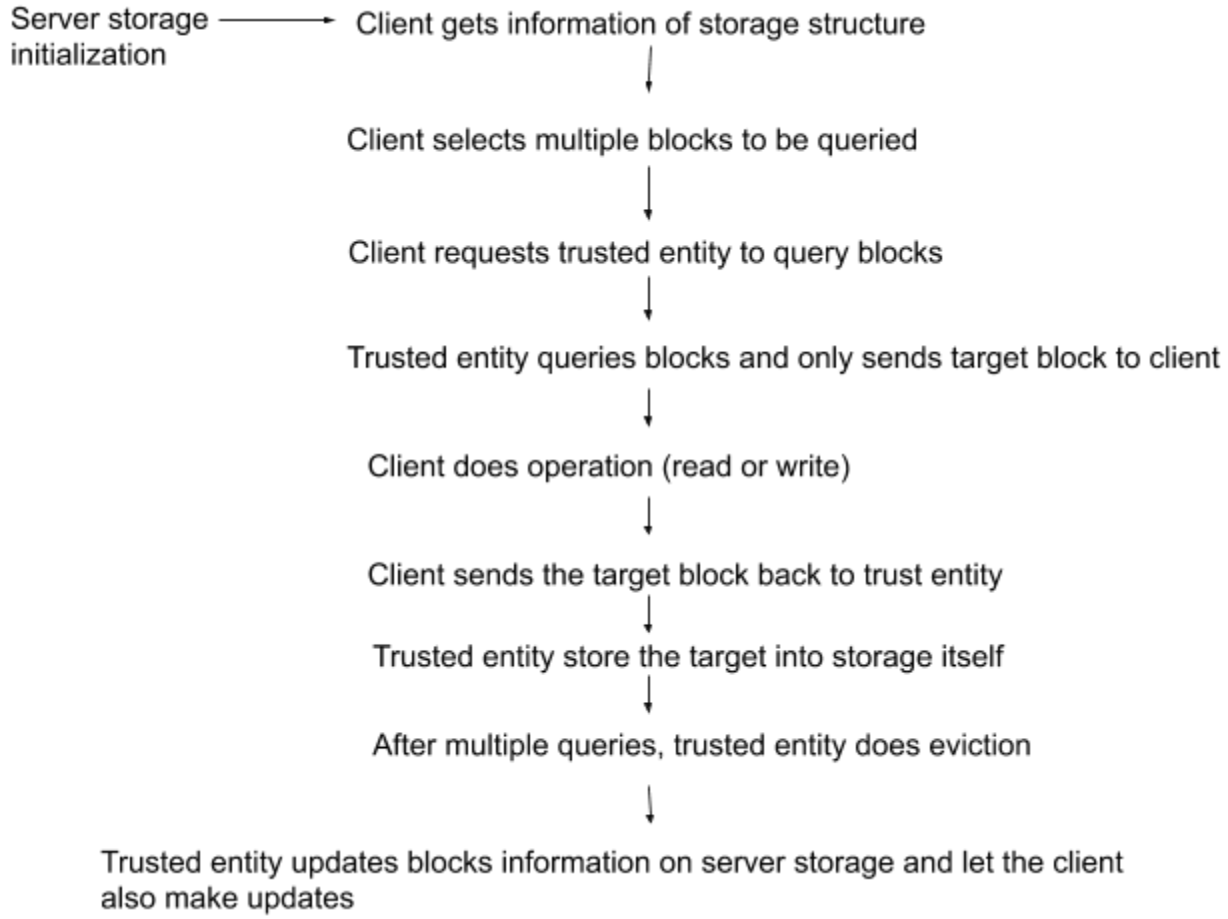
During the eviction process, the trusted entity needs to update target blocks' position and store these information because after eviction to be done, the trusted entity has to notify the client the updated target blocks' position so that the client can find them correctly if needed in the future.

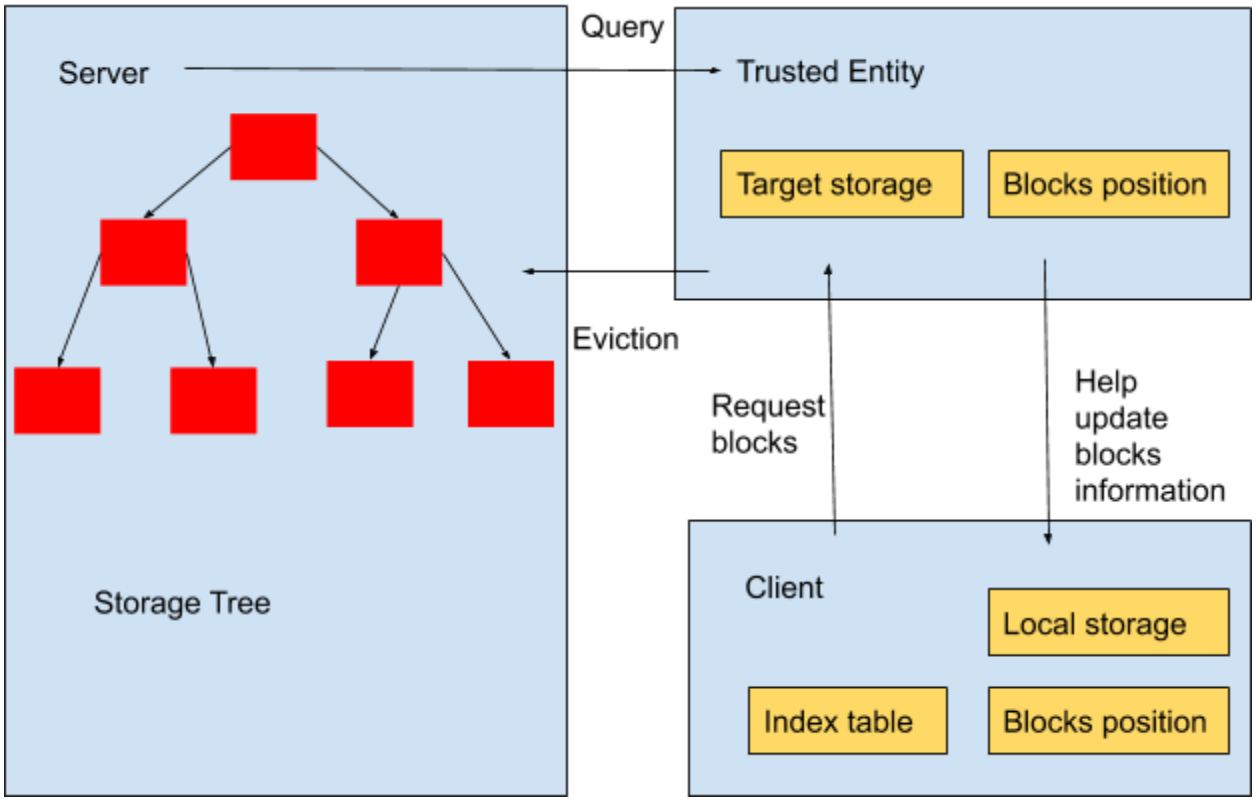
Finally, the client will update information responded from the trusted entity and the whole process of our ORAM protocol is shown as a flowchart in the below figure. In order to make it clear, there will be another figure showing the proposed system architecture after the flowchart followed by the flowchart.

In this system, we do not consider the trusted entity's job duty and our assumption is 'Custom first'. We need to help the client reduce wasted operations as much as possible but also at the same time, we keep the system safe enough to protect

clients' privacy. From the process shown, we can see no communication exactly exists between clients and server, which meets our safety concern requirements.

We assume that the trust entity can meet all requirements including enough space and fast computation power to run tasks that are on a large scale.



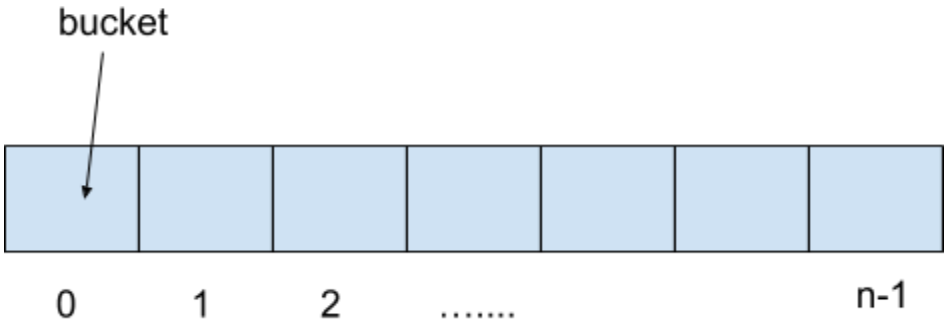


Implementation:

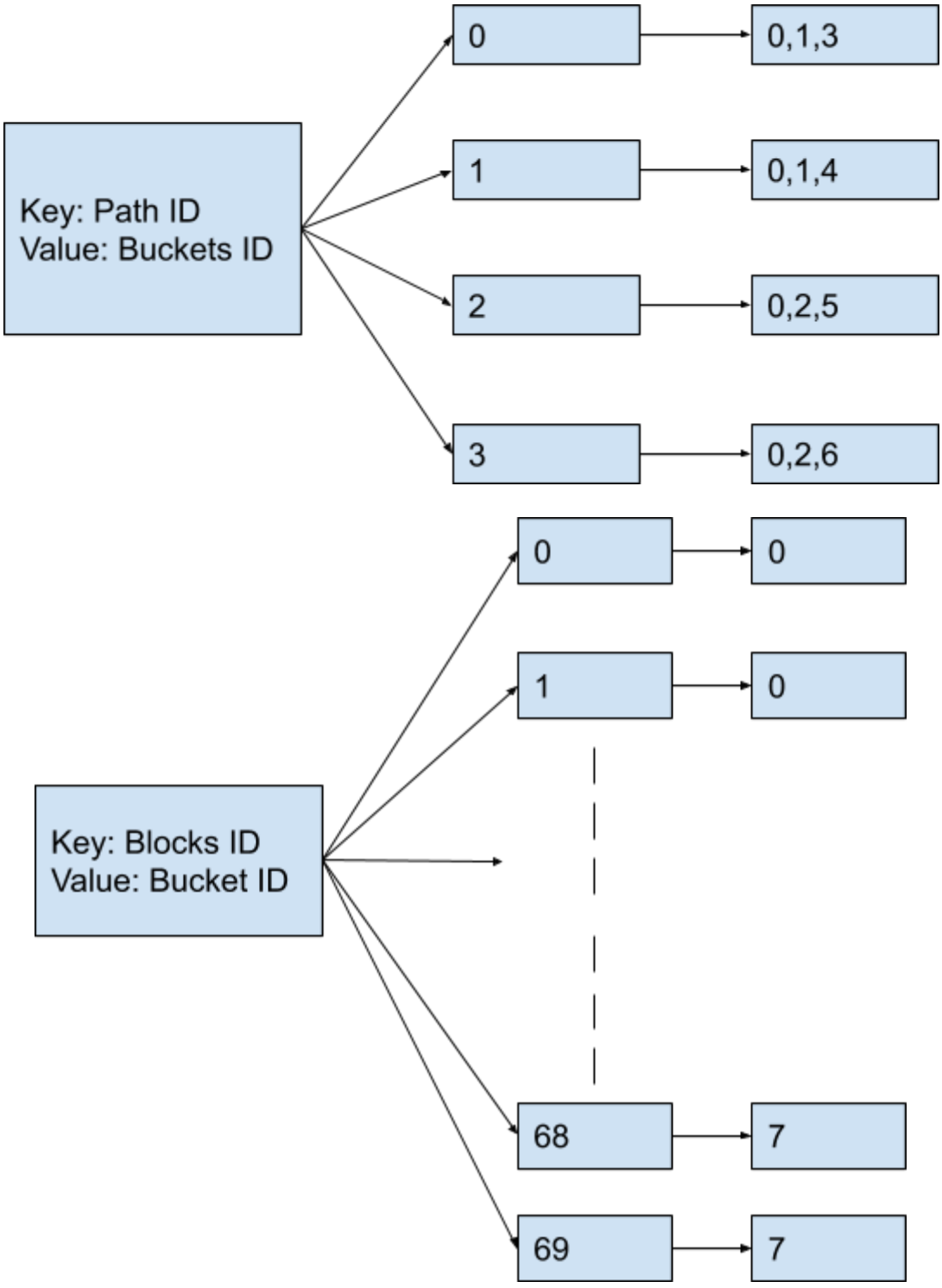
We focused on implementation except for building theoretical protocol. We use Java to implement this ORAM system with a trusted entity emulated. Here we mainly discuss the data structure with some examples we use for three entities to use to help implementation.

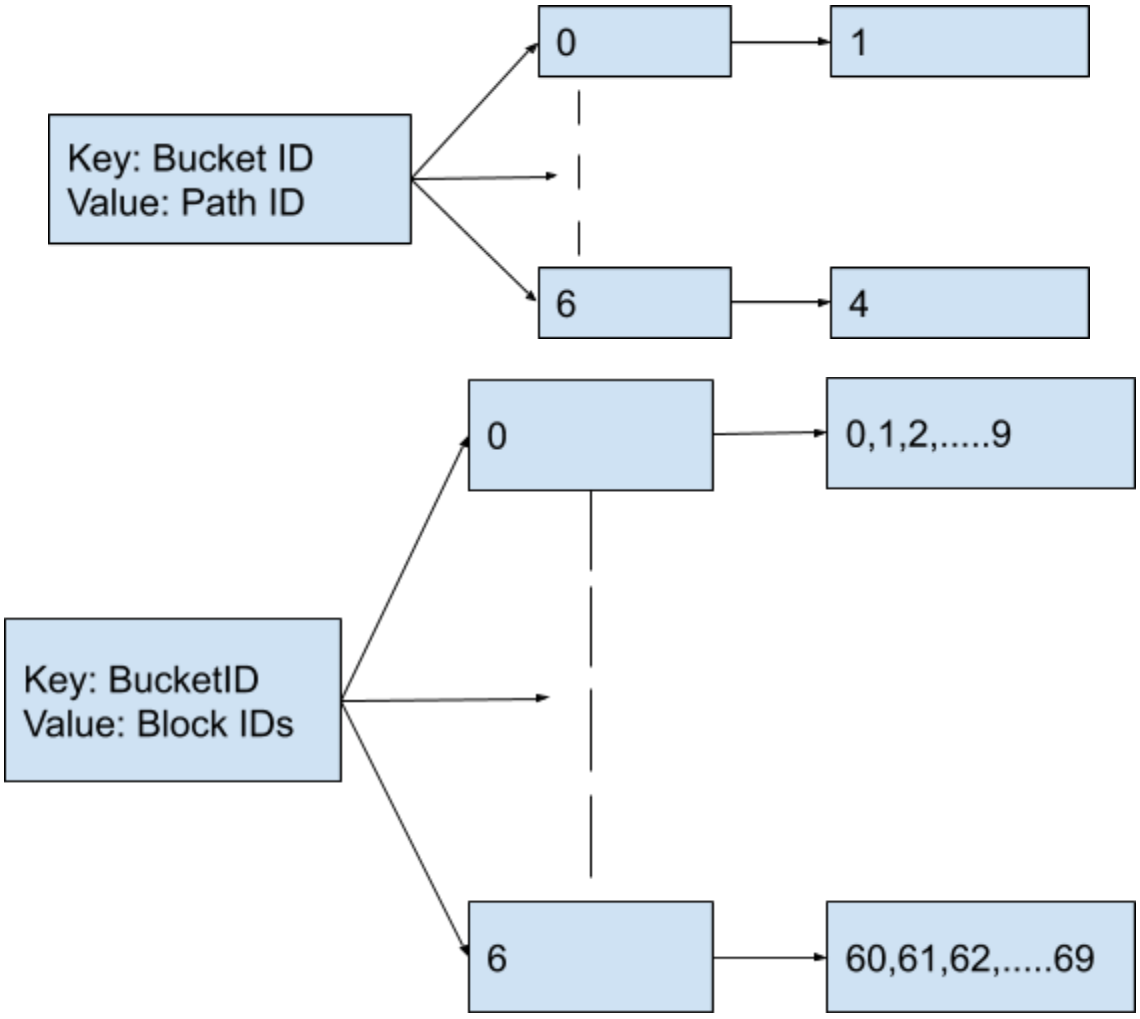
On the server side, there's not many things we need to do. We only need to store blocks in every bucket. Here, we use array implementation to implement tree structure because of lower space cost. In the real world, identically, on the server side, there should be nothing at the beginning, and clients continuously store blocks into the server but in this implementation, in order to make the system convenient and test the algorithm easily, we initially store blocks in the server storage.

Since the array is 0-index based data structure, so Bucket {0} is the root of the tree storage. Assume we are trying to implement a binary tree structure, then if the root bucket ID is n, the left child bucket ID will be 2n+1, and the right child bucket ID will be 2n+2. The basic tree storage structure with n buckets is shown as below clearly.



On the client side, in order to improve amortized efficiency, we need to do preprocessing work to help the client get information about tree storage structure directly. Also, we need to pay attention that the client needs to get all information as efficiently as possible including (1) If we know the path ID, the client can find all buckets on this path. (2) If we know the block ID, the client can find which bucket does this block locate at. (3) if we know the bucket ID, the client can directly find which path it is located on. (4) If we know the bucket ID, the client directly can find which blocks are located at it. In order to meet these four requirements, we have to implement four storage data structures to save these information. First, we create a map whose key represents the path ID and the value is the list of bucket IDs. Second, we need to create another map whose key represents the block ID and the value is corresponding bucketIDs. Third, we need to create one more map whose key represents the bucket ID and the value is the corresponding path ID. Finally, we need to build a map whose key represents each bucket ID and the value is the list of corresponding block IDs. Before the system starts, we build this storage in the client and afterward, all changes, add or remove operation will only cost O(1) which makes our system much more efficient compared to without preprocessing. Figures below show an example of client storage based on 7 buckets stored in the server with binary tree structure and each bucket has 10 blocks.


Key: Path ID
Value: Buckets ID
0
0,1,3
1
0,1,4
2
0,2,5
3
0,2,6
Key: Blocks ID
Value: Bucket ID
0
0
1
0
68
7
69
7



On the trusted entity side, since the trusted entity can directly access server storage, it only needs the information of path information developed by the client. Therefore, the trusted entity only needs to create a map whose key represents the path ID and the value is the list of bucket IDs before system running.

We use socket programming to build communications between the client and the trusted entity. There's no communication between the client and the server because of our system design. Also, There is no communication between the server and the trusted entity, because they are on the same side in our implementation. In the real world, both of them are on the same side either theoretically.

System Analysis:

To analyze the system performance, we both analyze the space cost and the time cost for each function done by all three entities. Then we can get a summary of the

performance. Also, we analyze how this system successfully hides clients data access patterns by examples.

We assume in our system, the storage tree height is log(N), which N is the total number of blocks N. For a client's preprocessing, the client has to build three maps which time complexity is O(N) because it has to check every block actually. However, this only happens once before the system starts. Therefore, the amortized time complexity of preprocessing is O(N/K) which K represents how many times the client queries the target block. The space complexity to store this information is O(N) exactly, however, for block storage on the client side, it's O(1) because the client only accesses one target each time and sends it back to the trusted entity. While we consider the client's query processing, we notice that the client has to check all blocks in one particular path, which means the time complexity for each query is log(N). Since the client does not need to access all blocks in one path, the space complexity for query is much less than log(N), which is much better than traditional ORAM based on binary tree storage structure.

On the server side, it only builds server storage based on array implementation. So it's obvious that time complexity and space complexity is O(N).

On the trusted entity side, the time complexity of the eviction algorithm is actually O(K) which K represents how many times the client makes requests. During accessing blocks on the server storage, the time complexity and space complexity are much less than log(N) because the number of required blocks is much less than log(N).

Besides time and space complexity analysis, we also need to show the obliviousness of the query and eviction process.

(1) Obliviousness in the query path selection. Each block is randomly located at server storage. Also, after multiple queries, their new located path is randomly selected which means this process is independent of the client's data access pattern.

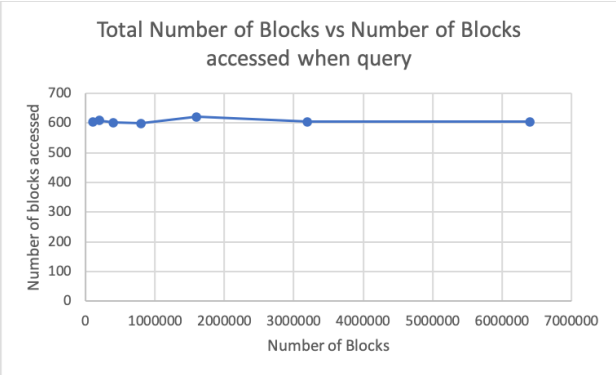
(2) Obliviousness in the block access process. For each bucket on the path, we need to select at least one block based on our query algorithm, which promises that the server won't get enough information of the client's data access pattern.

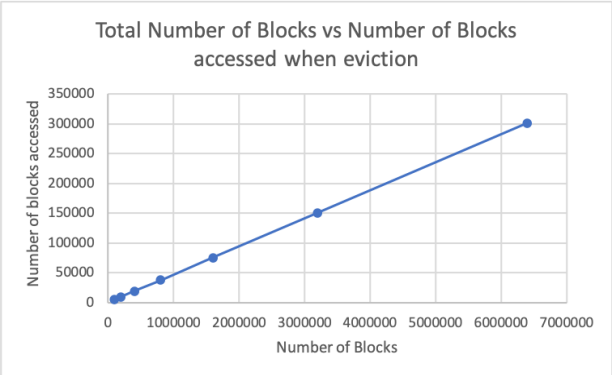
(3) Obliviousness in the eviction process. For each target block, we need to evict it into a new random path from root to leaf. Since it's encrypted and the size of every block is the same, the server does not know the details of the block so the client's data access pattern will not be leaked to the server.

Results Analysis:

We analyze the result of our implementation based on different attributes we are paying attention to. First, we tried to see how the total number of real blocks affects our system performance. We assume a client makes 100 queries, the height of the tree

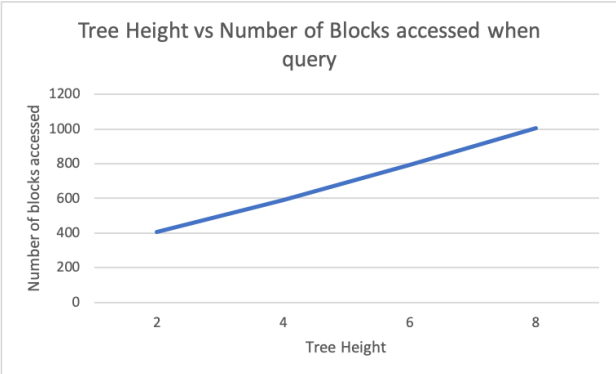
storage is 4, also, the degree of the tree storage is 4. Figures below show the query delay and the eviction delay based on the total number of real blocks.

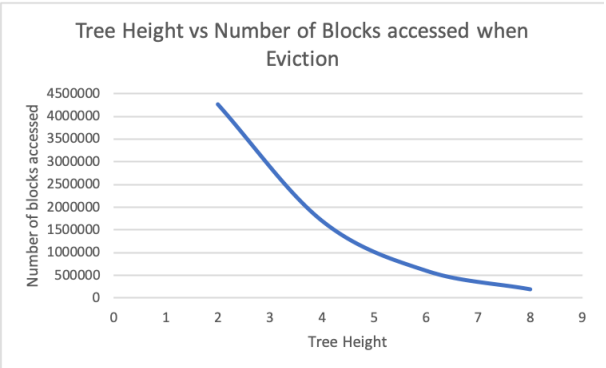




We can see query delay won't change a lot. This is because we make the structure of tree storage fixed. In the query process, the client needs to traverse a path of the tree and nearly query an average of 2 blocks from each bucket. In the eviction process, the trusted entity needs to traverse all blocks in a path and with an increasing number of blocks, the trust entity has to access more blocks.

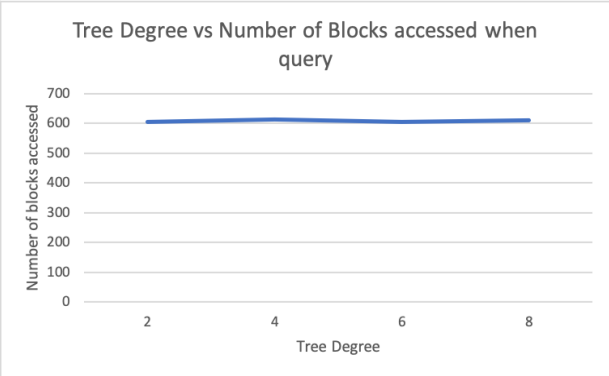
Second, we tried to see how the height of the tree affects our system performance such as query delay and eviction delay. We still assume a client makes 100 queries, the degree of tree storage is 2 which means it is a binary tree structure storage. The total number of blocks we set is 6400000 still because we hope our system can run successfully on a large scale. Figures below show the query delay and the eviction delay based on the tree height.

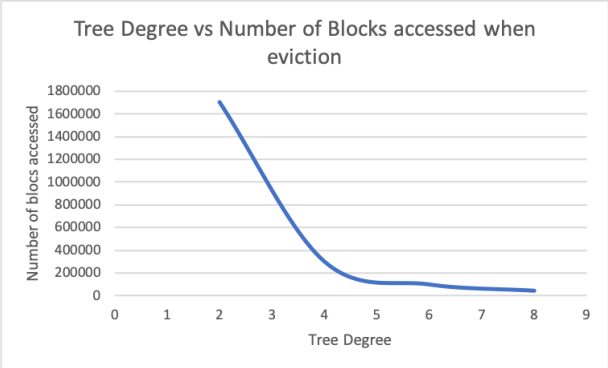




Exactly, once the height of the tree storage increases, the query delay increases. This is because per query, the client will query an average of 2 blocks per bucket in one path, and if the path is longer, more blocks the client will query. However, eviction delay decreases while the height of the tree increases. This is because per eviction, the trusted entity needs to access every block in every bucket in the target path. Even the height of the tree increases, but the number of blocks in every bucket reduces more.

Third, we tried to check how the degree of the tree affects our system performance. We still implemented this system with totally 6400000 real blocks, and we fixed the height of the tree as 4. Figures below show the result of our analysis.





Also, while we increase the degree of the tree, query delay will not change. This is because since the height of the tree is fixed, while the client queries an average of 2 blocks in each bucket, the number of blocks accessed will not change. The eviction delay decreases with the increasing of the degree of the tree because with higher tree degree, the number of blocks stored in each bucket will decrease so the trust entity will access fewer blocks during eviction.

Future Work:

To achieve this implementation, we tried to implement the trust entity by ourselves. In the future, we might try some real industrial product like Intel SGX to achieve a more realistic system which can be used in some practical systems.

Also in our implementation, we set lots of dummy blocks to make our system work properly even with lots of queries. However, more dummy blocks, more space cost that the server and the client might need to suffer from. In the future, we will try to find a balance of that with proper less dummy blocks, but still make our system work properly with a large scale of clients' demands.

Conclusion:

In this report, we discussed the development of Oblivious RAM and introduced our implementation of an ORAM system with three entities. Oblivious RAM is used to protect clients' data access pattern. We designed this proposed system based on reducing communication cost, reducing client storage and ensuring security concerns. Also, because lots of ORAM are just proposed but not implemented due to lots of specific reasons, we focused a lot on implementing this system in Java and analyzing its correctness. We tested our system on a very large scale (6400000 blocks totally) and it works the same as we theoretically proved before we implemented it.

We learnt from lots of works proposed by former researchers such as tree structure storage design, random eviction algorithm and methods of analyzing security concerns. In the future, we will try to implement this system with practical industrial products like Intel SGX and also optimize the system to improve the performance as much as possible.

References:

1. O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In STOC, 1987.
2. R. Ostrovsky. Efficient computation on oblivious rams. In STOC, 1990.
3. Stefanov, Emil, et al. "Path ORAM: an extremely simple oblivious RAM protocol." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013.
4. Zhang, Jinsheng, Wensheng Zhang, and Daji Qiao. "S-oram: A segmentation-based oblivious ram." *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 2014.
5. Ma, Qiumao, and Wensheng Zhang. "Efficient and Accountable Oblivious Cloud Storage with Three Servers." *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2019.
6. Stefanov, Emil, Elaine Shi, and Dawn Song. "Towards practical oblivious RAM." *arXiv preprint arXiv:1106.3652* (2011).
7. Shi, Elaine, et al. "Oblivious RAM with O ((logN) 3) worst-case cost." *International Conference on The Theory and Application of Cryptology and Information Security*. Springer, Berlin, Heidelberg, 2011.
8. Lu, Steve, and Rafail Ostrovsky. "Distributed oblivious RAM for secure two-party computation." *Theory of Cryptography Conference*. Springer, Berlin, Heidelberg, 2013.
9. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In SODA, 2012.