Lecture Notes for Cryptographic Computing
8. Oblivious RAM

Lecturers: Claudio Orlandi, Peter Scholl, Aarhus University

October 23, 2023

1 Oblivious RAM

This week, we will look at a different aspect of secure computing called *oblivious RAM* [**?**]. Unlike most of the other techniques covered previously, here the focus is not on how to hide private data itself, but instead, how to hide the *access pattern* that can be observed from making queries to private data.

Oblivious RAM considers a setting between a *client* and a *server*, where the client outsources the storage of a database to the server. We represent the database as an array, D, which the client wishes to access by performing read or write operations. To help preserve privacy, the client may use symmetric-key encryption to hide the contents of D from the server.

ORAM for secure hardware. Another use-case for ORAM, which is often discussed in the literature, is the setting of secure hardware. Here, instead of a client accessing data stored at a server, we consider a secure processor, which accesses an untrusted memory resource. This is relevant, for instance, in a *secure enclave* such as Intel SGX, which aims to isolate the execution of code and sensitive data from an untrusted user. Any program running in a secure enclave that requires a large amount of memory will typically leak its access pattern to this memory; oblivious RAM can help to hide the access pattern.

ORAM for secure multi-party computation. Finally, Oblivious RAM can also be used to improve the efficienc of certain types of operations in secure computation. Standard MPC protocols such as BeDOZa require the function to be expressed as a circuit, however, this means that an operation such as lookup to an array, which is normally $O(1)$, uses $O(n)$ gates when written as a circuit. By combining ORAM with MPC, we can obtain better efficiency for using arrays and other data structures inside secure computation. We will not explore this further in the course, but for further reading, see e.g. [**?**].

1.1 Access Pattern Leakage

Even when D is stored encrypted, if the *access pattern* of read and write operations that the client makes to D can be observed, then this may still leak sensitive information to the server. For instance, suppose the client is a medical researcher making queries to a database of patient records, for the purpose of studying a particular disease. If the server can observe which patients' records are accessed, this could allows the server to infer (with reasonable probability) that these patients are likely to have the disease. This type of attack has been shown to be feasible in various scenarios, such as performing keyword searches to a database of encrypted emails [**?**].

1.2 ORAM Definition

In an ORAM, we first initialize the client and server's state, with the security parameter λ and the contents of the initial database D. Note that after initialization, the server will not storing D directly, instead, it has an *ORAM database* ORAM that encodes D in some special way. Then, the client and server engage in Read and Write protocols as needed.

- $\mathsf{Init}(1^\lambda, D) \to (K, \mathsf{ORAM})$: initializes the ORAM, outputting the client's secret state K and server's storage ORAM.
- $\mathsf{Read}((K, i), \mathsf{ORAM}) \to ((K, D[i]), \mathsf{ORAM})$: a two-party protocol with input (K, i) from the client and ORAM from the server, which updates each party's state and outputs $D[i]$ to the client.
- $\mathsf{Write}((K, i, x), \mathsf{ORAM}) \to (K, \mathsf{ORAM})$: a two-party protocol with input (K, i, x) from the client and ORAM from the server, which updates the value of $D[i]$ to contain x.

The correctness property of an Oblivious RAM scheme simply requires that the Read and Write operations behave as expected, by reading and modifying the elements of D.

To define security of ORAM, let A be some sequence of $\mathsf{Read}/\mathsf{Write}$ operations. Let $\mathsf{View}_S(D, A)$ be the view of a corrupt server in the ORAM protocol, where first Init is run on the database D to give the client and server their state; then, the two parties run the Read or Write protocol for each operation in A. Note that View_S contains the initial state of the server, together with all its randomness and all messages received by the client.

Definition 1. *We say that an ORAM scheme is oblivious, if for all databases D and all access patterns A_0, A_1 of the same length:*

$$\mathsf{View}_S(D, A_0) \cong \mathsf{View}_S(D, A_1)$$

The above definition guarantees that any two possible access patterns of the same length are indistinguishable to the server, when one of them is run through the ORAM protocol. Just like IND-CPA security for encryption, this ensures that all information about the secret access pattern remains hidden.

Exercise 1.

The ORAM definition requires that the server cannot distinguish whether the client is *reading* memory location i, or *writing to* the same location.
Suppose we have some ORAM protocol Π, where you *can* distinguish between read or write operations, but Π still hides the *locations* which were accessed (i.e. the indices i). Can you use Π to construct another ORAM which satisfies the stronger definition?

Efficiency Metrics. To measure the cost of an ORAM, we can consider the *client storage cost*, as well as the *server storage overhead* and the *bandwidth overhead* for the read/write protocols. For instance, if the server's storage ORAM is 20x larger than D, then we say the server storage overhead is a factor of 20. Or, if each entry of the original D has size 1 kB, but a Read or Write protocol involves 32 kB of communication, we have a bandwidth overhead of 32x.

1.3 Naive ORAM Solutions

The trivial solution to constructing ORAM is to have the server send the entire database D to the client, for every access query. Then, the client decrypts D, locally performs the desired operation, re-encrypts the entire database and sends it back to the server. Of course, this solution is prohibitively expensive if D is reasonably large.

Another possibility for constructing ORAM, which is explored in the exercise below, is to use fully homomorphic encryption.

Exercise 2.

Describe how to use fully homomorphic encryption to build an Oblivious RAM protocol.
What do you think about the efficiency of your protocol, in terms of storage/bandwidth overheads, and computational costs?

2 Constructing Square-Root ORAM

A natural first attempt to build ORAM is to store a *randomly shuffled* version of the database at the server, to break the association between the actual access operations and the indices that are seen by the server. Concretely, we initialize the ORAM as follows:

Init(D):

1. The client chooses a random permutation π on $\{0, \ldots, n-1\}$ (to avoid $O(n)$ storage costs, this can be sampled using a pseudorandom function; see Exercise 3)
2. Define the permuted database $\widetilde{D}$, where
$$\widetilde{D}[\pi(i)] = D[i], \quad \text{for } i = 0, \ldots, n-1$$
3. Output the client state π, and the server storage ORAM $:= \widetilde{D}$

Now, to access element i of D, the client will send $\pi(i)$ to the server, who responds with $\widetilde{D}[\pi(i)]$. The index i is called the *virtual location* of $D[i]$, while $\pi(i)$ is its *physical location* at the server.

If no index is queried more than once, this clearly hides the access pattern: the server only sees a sequence of non-repeating, random locations. However, with repeated queries, the server will observe that the same memory location is being accessed again. Therefore, this does not satisfy Definition 1; instead, we can think of it as a *one-time secure* ORAM. To upgrade this to full security, there are two key modifications to make.

Exercise 3.

Storing the permutation π requires the client to use $O(n)$ storage. Can you think of an alternative way to shuffle the items in D, while storing only a key for a pseudorandom function?
Hint :

Remark 1 (Small-domain PRPs: for extra interest only)**.** *In the exercise above, you may have noticed that one tool that could solve this problem is a* pseudorandom permutation on small domains*. Note that typically, PRPs such as block ciphers have a very large domain (e.g.* 2^{128} *for AES-128), whereas here we want a secure PRP with a domain of size* n*. These have been studied for the application of* format-preserving encryption*, which can e.g. encrypt a credit card number, giving a ciphertext which also looks like a credit-card number. The best-known constructions are often based on techniques inspired by card shuffling, such as swap-or-not [?] or the mix-and-cut shuffle [?].*

2.1 Step 1: the Stash

To help allow repeated queries, the client will locally store a *stash*, an array of temporary storage for up to T items (for some parameter T). Now, whenever the client wants to access element i, it first checks whether $(i, D[i])$ is in the stash. If the stash lookup fails, the client sends $\pi(i)$ to the server to retrieve $D[i]$, before adding $(i, D[i])$ to the stash.

How oblivious is this? With a stash, the client now never queries the same index twice to the server. However, note that when querying, there is now a different interaction pattern depending on whether the element was found in the stash or in server-side storage: the client only sends a request to the server if i is not in the stash. In this case, even though there is no need for communication with the server, when combined with other queries in an application this may lead to a kind of side-channel attack, where the server can observe the pattern of requests made by the client, and try to learn whether the same locations are being accessed or not.

To mitigate this, during initialization, we append T additional *dummy items* to the database D, in locations $n, \dots, n+T-1$, before shuffling. The server's ORAM storage is now the shuffled array $\widetilde{D}$, of size $n+T$. Then, whenever the client's query is found in the stash, the client performs an access to the next unused dummy item (again, mapped to a random index via π). This ensures that, in every read operation, the server sees the same interaction pattern: a single query to a random index of $\widetilde{D}$, which was not queried before.

2.2 Step 2: Reshuffling

At this point, the client can obliviously perform arbitrary queries to the shuffled database, until the stash of size T fills up. At this point, the client can no longer make repeated queries to items which aren't in the stash, since the server would observe that the same location is being accessed.

To be able to continue, we perform a *reshuffling* stage, which essentially re-initializes the entire data structure, allowing the client to start with an empty stash. This an expensive operation, however, we will only need do it once every T accesses. To reshuffle, if the client has $O(n)$ local storage space, then it can simply download the entire database $\widetilde{D}$, decrypt it to recover D, then sample a new permutation π to setup a new ORAM at the server.

If we instead want to keep the client storage small, then reshuffling is more challenging. The difficulty is that we need to *obliviously* reorder the database, so that the server learns nothing about the new permutation that maps virtual indices of D to their physical locations, while maintaining only small storage at the client. For this, we use a tool called a *sorting network*.

Sorting Networks. A sorting network is a way to sort an array in a *data-oblivious* manner, using a special type of circuit. The circuit has n inputs and n outputs, and consists entirely of *comparison gates*, which take two inputs (x, y), and output (x, y) if $x \leq y$, otherwise outputting (y, x). A key property of sorting networks is that the wiring and gate structure of the circuit is *independent of the inputs*; so, the pair of elements that will compared at any given step can be determined without knowing the underlying data.

Practical sorting networks can be built with a circuit of $O(n \log^2 n)$ comparison gates, using Batcher's odd-even mergesort.[1] An example for $n = 8$ is in Fig. 1.

$\mathsf{Reshuffle}$ **protocol:**

1. The client picks a new random permutation ρ on $\{0, \dots, N+T-1\}$
2. The client and server jointly evaluate a sorting network on $\widetilde{D}$; for each successive comparison gate, with inputs $\widetilde{D}[j]$ and $\widetilde{D}[k]$:

[1]The AKS sorting network [?] has size $O(n \log n)$, however, it is estimated to have a hidden constant of around 2^{100}.

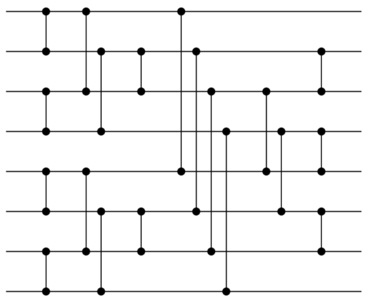

Figure 1: Visualization of the odd-even sorting network for $N = 8$

   (a) The server sends $\widetilde{D}[j], \widetilde{D}[k]$ to the client
   (b) The client decrypts these to recover data items (i_1, x_1) and (i_2, x_2)
   (c) If $\rho(i_1) \le \rho(i_2)$, the client swaps the two items. It then re-encrypts them and sends them back to the server.

3. After shuffling, the server's ORAM array contains a freshly shuffled database according to ρ.

Complete Read Protocol. Putting all of the above together, we can formalize the Read protocol as follows. The client maintains a counter $\mathsf{ctr} := 0$, to keep track of how many accesses have been performed. To read a virtual location i, the client does as follows:

1. If i is in the stash:
   (a) The client retrieves $x = D[i]$ from the stash
   (b) Compute $j = \pi(N + \mathsf{ctr})$ and send j to the server
   (c) The server sends back $\widetilde{D}[j]$
2. Else:
   (a) The client computes $j = \pi(i)$ and sends j to the server
   (b) The server sends back $\widetilde{D}[j]$
   (c) The client decrypts $\widetilde{D}[j]$ to recover $(i, x = D[i])$
   (d) The client adds (i, x) to the stash
3. Set $\mathsf{ctr} := \mathsf{ctr} + 1$
4. If $\mathsf{ctr} = T$:
   (a) Run Reshuffle to re-initialize π and $\widetilde{D}$
   (b) The client clears the stash
5. Output x

Handling Write queries. So far, we have focussed on Read operations. In the following exercise, you will extend the Read algorithm to support write queries. This is mostly straightforward, but requires some care to handle any potential inconsistencies between the stash and server storage.

Exercise 4.

Describe the complete protocol for Write queries, and argue its correctness and obliviousness.

Efficiency. We now analyze the overheads of the square-root ORAM protocol. Throughout the protocol, the server stores an array containing $n + T$ items, which is less than 2x storage overhead since $T < n$. The client, on the other hand, has an $O(T)$ storage overhead to store the stash. Each Read query involves one query to the server, except for every T-th query, where we run the Reshuffle protocol. This involves a further $O(n \log^2 n)$ communication between the client and server.

Overall, while the *worst-case* bandwidth overhead for a single query is $O(n \log^2 n)$, on *average*, we get an amortized overhead of $O(n \log^2 n/T)$. Choosing $T = O(\sqrt{n})$, the amortized bandwidth overhead becomes $O(\sqrt{n} \log^2 n)$, while the client storage cost is $O(\sqrt{n})$.

3 Tree-Based ORAM

To do better than $O(\sqrt{n})$ overhead, a simple and relatively practical way to build ORAM is with a tree-based construction. Here, we will use the construction by Chung and Pass [?], as it has a relatively simple security analysis. We start with a construction which reduces the storage costs of the client by a constant factor from the trivial solution. Then, we show how to use recursion to obtain a solution which requires only $O(\log n)$ client memory.

3.1 Basic Construction

Suppose the database D consists of n blocks of size B bits each, where $B > c \log n$ for some constant c. The main components of the ORAM are as follows.

- Client storage: a *position map*, Pos, which stores a randomly chosen index $\ell_i \in \{0, \ldots, n-1\}$ for each index $i \in \{0, \ldots, n-1\}$. That is, $\mathsf{Pos}[i] = \ell_i$, for all i.
- Server-side storage: the database is kept in a *complete binary tree* of depth $\log n$, with n leaves. Each node in the tree consists of a *bucket* with up to K data items of the form (i, ℓ_i, v_i), where
  - $v_i = D[i]$ is the i-th memory value.
  - $\ell_i = \mathsf{Pos}[i]$ is an index of a leaf node associated with index i.

Main invariant: Any tuple (i, ℓ_i, v_i) stored in the tree lies somewhere on the path between the root of the tree and leaf node ℓ_i.

To initialize the ORAM, the tree can be initialized by randomly sampling a permutation to define Pos, so that each leaf bucket in the tree is assigned exactly one data item. The ORAM's tree structure is illustrated in Fig. 2.

Read algorithm. The Read algorithm proceeds in two stages: the *fetch* algorithm retrieves the corresponding item from the server, and then the *evict* algorithm is used to re-balance the tree, preventing buckets from overflowing.

Fetch. First, the client looks up $\ell_i = \mathsf{Pos}[i]$, and requests the entire path of buckets from the root to leaf ℓ_i from the server. The client removes the item (i, ℓ_i, v_i), then picks a new leaf ℓ'_i at random and adds (i, ℓ'_i, v_i) to the root bucket. Finally, it sends back the new path to the server.

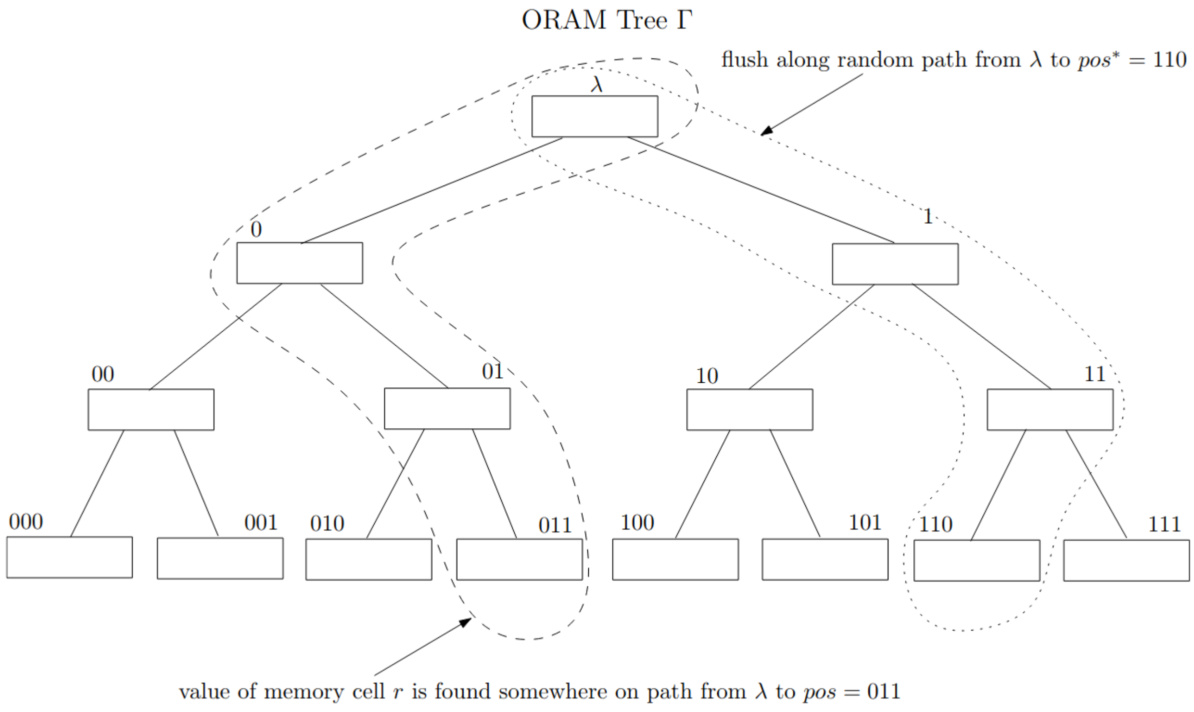
ORAM Tree Γ
flush along random path from λ to $pos^* = 110$
λ
0
1
00
01
10
11
000
001
010
011
100
101
110
111
value of memory cell r is found somewhere on path from λ to $pos = 011$

Figure 2: Tree-based ORAM structure. Figure from [CP13]

Evict. Next, the client picks a random leaf ℓ^* and requests the path to ℓ^*. Now perform the following "push-down" operation: for each item $V = (i, \ell_i, v_i)$ in the path, push V as far down the path as possible, while maintaining the invariant that it is still on the path to ℓ_i.

The client then encrypts and sends back the updated buckets on the path to ℓ^*. If any bucket overflows, the client aborts.

Write algorithm. As with the previous construction, the Write algorithm is a fairly straightforward modification of Read, so left as an exercise.

Exercise 5.

Describe the Write algorithm for the tree-based ORAM scheme.

Obliviousness. It is easy to see that, as long as none of the buckets overflow, the access pattern seen by the server is independent of the client's virtual memory accesses. This is because, in each fetch or eviction procedure, the client always requests an independent, random path of the tree.

3.1.1 Overflow analysis.

The challenging part of this construction is to bound the probability that no bucket in the tree overflows. Consider the following dart game:

- You have both black and white darts
- In each round of the game, first throw a black dart, and then a white dart. Each dart hits the target with probability p
- When K darts have hit the target, the game ends

You "win" the game if at the end, all of the K darts that hit the target are black.

Exercise 6. (Hard: use the hints)

Show that the probability of winning the dart game is no more than 2^{-K}.
Hint 1:

Hint 2:

The dart game relates to tree-based ORAM in the following way. Consider any internal (i.e. non-leaf) node γ in the tree. For the bucket in node γ to overflow, it must first contain K data items, where each item is of the form (i, ℓ_i, v_i), where γ is a *prefix* of ℓ_i (since γ lies on the path to ℓ_i). Now, since every item in bucket γ is either assigned to a leaf in the left sub-tree or the right sub-tree of γ, at least $K/2$ of these items are assigned to a leaf with prefix $\gamma\|b$, for some $b \in \{0, 1\}$.

Think of black darts hitting the target as memory items with leaves with prefix $\gamma\|b$, and the white darts hitting the target as leaves chosen during eviction, also with prefix $\gamma\|b$. Now, for there to be $K/2$ items with prefix $\gamma\|b$ in the bucket, we must have assigned $K/2$ random leaves with this prefix, *without* performing an eviction on the path towards any leaf with the same prefix. Since both eviction leaves and memory item leaves are picked at random, they each have the same probability of having this prefix; call this probability p. Therefore, the probability that bucket γ overflows in any given access is no more than the probability of winning a dart game with $K/2$ darts, which is $2^{-K/2}$.

By a union bound over all internal nodes and the number of accesses, T, the overall probability of an internal node overflowing is no more than $nT2^{-K/2}$.

Finally, we also need to analyze the overflow probability of a leaf node. We will omit this in these notes, but instead refer to the paper [**?**], where it is shown that by applying a Chernoff bound, this probability can also be upper-bounded by $2^{-K/2}$.

3.2 Using Recursion

In the basic tree-based construction, the client stores a position map of n indices, that is, $n \log n$ bits. If the block size of the original data is larger than $\log n$, then this gives a small amount of compression, but is still $O(n)$.

To reduce the client's storage, we will apply the construction recursively. Let Pos be the position map in the basic construction. Define a new database D_1 of size $n/2$, which encodes the original position map as

$$D_1[i] = (\mathsf{Pos}[2i], \mathsf{Pos}[2i+1]), \qquad \text{for } i = 0, \dots, n/2$$

Now, instead of having the client store Pos locally, we will create a *second* ORAM that stores D_1. Whenever the client needs to do a lookup to $\mathsf{Pos}[j]$, it will first use the second ORAM to retrieve $D_1[\lfloor j/2 \rfloor]$, and then extract one of the two indices. This second ORAM will have its own position map, Pos_1, stored by the client, but this only contains $n/2$ indices.

Assuming n is a power of 2, we can repeat the above process $\ell = \log_2 n$ times, obtaining a sequence of ORAMs on databases $D_0, \dots, D_{\ell-1}$, where D_0 is the original database, and each D_i for $i > 0$ is of size $n/2^i$, and encodes the position map for the ORAM that stores D_{i-1}. After this, the client only needs to store the final position map $\mathsf{Pos}_{\ell-1}$, which has size two.

Efficiency Analysis. Using recursion, the client only needs to store a single position map of constant size. To retrieve the paths during access to each of the ORAMs, though, the client needs $O(\log n)$ memory. In the exercise below, we analyze the bandwidth and server-side storage overheads

Exercise 7.

Calculate the asymptotic bandwidth and server-side storage of the recursive construction, as well as the round complexity, in terms of the database size n and bucket size K.
Can you think of any optimizations that might simplify and/or optimize the consruction in some way?