2.3 Privacy: Oblivious RAM

Elaine Shi

The setting: you have a large amount of private data (e.g., your genomic data) stored on an untrusted server. While standard encryption techniques allow the client to hide the contents of the data from the server, the server can still observe access patterns to the data. Through such access patterns, the server can potentially infer sensitive information about your private data. For example, through *frequency* and *co-occurrence* information, the server may be able to infer what genomic algorithm (e.g., medical test) is being executed on the genomic data. It is also helpful to think of access pattern leakage through a programming language perspective: for example, the following program has an `if`-branch dependent on secret inputs (e.g., think of the secret input as the last bit of a secret key) Thus by observing whether memory location `x` or `y` is accessed, one can infer which branch is taken.

```
if (s) {
  mem[x]
} else {
  mem[y]
}
```

Some well-known cryptography implementations are known to have such secret `if`-branches, e.g., the square-and-multiply algorithm (see Chapter 2.6.3 in Pass and shelat [PS10]) often used for efficient modular exponentiation.

Oblivious RAM: Problem Definitions

Oblivious RAM (ORAM), first proposed in the seminal work by Goldreich and Ostrovsky [GO96, Gol87], is a powerful cryptographic protocol that *provably* hides access patterns to sensitive data.

We would like to ensure a very strong notion of security. In particular, no information should be leaked about: 1) which data block is being accessed; 2) the age of the data block (i.e., when it was last accessed); 3) whether a single block is being requested repeatedly (i.e., frequency); 4) whether data blocks are often being accessed together (i.e., co-occurrence); or 5) whether each access is a read or a write.

We will now formally define ORAM and its security. An ORAM algorithm implements a logical memory abstraction. Henceforth, we refer to each atomic memory word as a memory *block*, and we use the notation N to denote the total number of blocks of the logical memory. We will also use N to denote the security parameter.

An ORAM algorithm (also referred to as the *client*) receives as inputs a sequence of *logical* requests where each logical request is of the form

$$(\texttt{read}, \mathsf{addr}) \text{ or } (\texttt{write}, \mathsf{addr}, \mathsf{data}).$$

Upon receiving each request, an ORAM algorithm may interact with a "memory" (also referred to as the *server*) to make a sequence of *physical* accesses where each physical access either reads or writes a physical location, at the end of which the ORAM algorithm returns an answer to the logical input request. The adversary (i.e., server) can observe this physical access sequence, but cannot directly observe the logical input requests or the private random coins used by the ORAM algorithm. In our lecture, we will consider ORAM schemes that ensure security even against computationally unbounded adversaries, but we allow that *with a very small (i.e., negligible in N) probability, the ORAM algorithm may return an incorrect result.*[1]

Given a logical request sequence X, the random variable $\mathsf{Addresses}(X)$ denotes the physical accesses (including the addresses as well as whether each access is a read or write) resulting from the ORAM algorithm for the input sequence X. We say that an ORAM scheme is secure iff for any N and for any two logical request sequences X_0 and X_1 of the same length,

$$\mathsf{Addresses}(X_0) = \mathsf{Addresses}(X_1),$$

where "=" denotes "identically distributed". Intuitively, our security definition requires that for any two *logical* request sequences, the ORAM's resulting *physical* access sequences will be indistinguishable against computationally unbounded adversaries.

Remark 2.3. Note that in our definition of $\mathsf{Addresses}$, i.e., what the adversary can observe, we did not include the contents of the memory blocks, only the physical addresses and whether each physical access is a read or write. In practice, we may use encryption to hide the contents of the blocks — here we simply assume secure encryption as given, and thus we only care about hiding the access patterns.

[1]Most academic papers on ORAM instead require that the ORAM scheme have perfect correctness, but allow a negligibly small probability of security failure.

Naïve Solutions

Naïve solution 1. One trivial solution is for the client to read all blocks from the server upon every logical request. Obviously this scheme leaks nothing but would be prohibitively expensive.

Naïve solution 2. Another trivial solution is for the client to store all blocks, and thus the client need not access the server to answer any memory request. But this defeats the numerous advantages of cloud outsourcing in the first place. *Henceforth, we require that client store only a small amount of blocks* (e.g., constant or polylogarithmic in N).

Naïve solution 3. Another naïve idea is to randomly permute all memory blocks through a secret permutation known only to the client. Whenever the client wishes to access a block, it will appear to the server to reside at a random location.

Indeed, this scheme gives a secure *one-time* ORAM scheme, i.e., it provides security if every block is accessed only once. However, if the client needs to access each block multiple times, then the access patterns will leak statistical information such as frequency (i.e., how often the same block is accessed) and co-occurrence (i.e., how likely two blocks are accessed together). As mentioned earlier, one can leverage such statistical information to infer sensitive secrets [IKK12].

Important observation. The above naïve solution 3 gives us the following useful insight: informally, if we want a "non-trivial" ORAM scheme, it appears that we may have to relocate a block after it is accessed — otherwise, if the next access to the same block goes back to the same location, we can thus leak statistical information. It helps to keep this observation in mind when we describe our ORAM scheme later.

Binary-Tree ORAM: Data Structures

We will learn about tree-based ORAMs, first proposed by Shi et al. [SCSL11]. Tree-based ORAMs provide a framework to construct very simple ORAM schemes: in our lecture, we will describe the original binary-tree ORAM by Shi et al. [SCSL11]; but in your lab, you will be implementing the Path ORAM scheme [SvDS$^+$13].

Server data structure. The server stores a binary tree, where each node is called a *bucket*, and each bucket is a finite array that can hold up to Z number of blocks — for now, think of Z as being relatively small (e.g., polylogarithmic in N); we will describe how to parametrize Z later. Some of the blocks stored by the server are *real*, other blocks are *dummy*. As will be clear later, these dummy blocks are introduced for security.

Main path invariant. The most important invariant is that at any point of time, each block is mapped to a random path in the tree (also referred to as the block's designated path), where a path begins from the root and ends at some leaf node — and thus a path can be specified by the corresponding leaf node's identifier. When a block is mapped to a path, it means that the block can legitimately reside anywhere along the path.

Imaginary position map. For the time being, we will rely on the following cheat (an assumption that we can get rid of later). We assume that the client can store a somewhat large position map that records the designated path of every block. In general, such a position map would require roughly $\Theta(N \log N)$ bits to store — but later we can recursively outsource the storage of the position map to the server by placing them in progressively smaller ORAMs.

Binary-Tree ORAM: Operations

We now describe how to access blocks in our ORAM scheme.

Fetching a block. Given how our data structures are set up, accessing a block is very easy: the client simply looks up its local position map, finds out on which path the block is residing, and then reads each and every block on the path. As long as the main invariant is respected, the client is guaranteed to find the desired block.

Remapping a block. Recall that earlier, we have gained the informal insight that whenever a block is accessed, it should relocate. Here, whenever we access a block, we must remap it to a randomly chosen new path — otherwise, we would end up going back to the same path if the block is requested again, thus leaking statistical information.

To remap the block, we choose a fresh new path, and update the client's position map to associate the new path with the block. We now would like to write this block back to the tree, to somewhere on the new path (and if the request is a `write` request, the block's contents are updated before being written back to the server). But doing this is tricky! It turns out that we cannot write the block back directly to the leaf bucket of the new path — since doing so would reveal which new path the block got assigned, this leaks information since if the next request asks for the same block, it would then go to this new path; otherwise most likely the next request will go to a different path. By a similar reasoning, we cannot write this block back to any internal nodes of the new path either, since writing to any internal node on the new path also leaks partial information about the new path.

It turns out that the only safe location to write the block back is to the root bucket! The root bucket resides on every path, and thus writing the block back to the root does not violate the main path invariant; and further, it does not leak any information about the new path.

Now this is great. Our idea thus is to write this block back to the root bucket. However, there is also an obvious problem! The root bucket has a capacity of Z, and if we keep writing blocks back to the root, soon enough the root bucket will overflow! Therefore, we now introduce a new procedure called *eviction* to cope with this problem.

Eviction. Eviction is a maintenance operation performed upon every data access to ensure that none of the buckets in the ORAM tree will ever overflow except with negligible in N failure probability. Note that if an overflow does happen, the block that leads to the overflow can get lost since there is no space to hold it on the server, and this can affect the correctness of our ORAM scheme. However, we will guarantee that such correctness failure happens only with negligible probability.

The high-level idea is very simple: whenever we can, we will try to move blocks in the tree closer to the leaves, to allow space to free up in smaller levels of the tree (i.e., levels closer to the root). There are a few important considerations when performing such eviction:

- Data movement during eviction must respect the main path invariant, i.e., each block can only be moved into buckets in which it can legitimately reside.

- Data movement during eviction must retain *obliviousness*, i.e., the physical locations accessed during eviction should be independent of the input requests to the ORAM.

Algorithm 1 Access(op, addr, data) where op = read or op = write

Assume: each block is of the form $(\mathsf{addr}, \mathsf{data}, l)$ where l denotes the block's current designated path.

```
1: $l^* \xleftarrow{\$} [1..N]$, $l \leftarrow \mathtt{position}[\mathsf{addr}]$, $\mathtt{position}[\mathsf{addr}] \leftarrow l^*$.
2: for each bucket from leaf $l$ to root do
3:     Scan bucket, and if $(\mathsf{addr}, \mathsf{data}_0, \_) \in \mathsf{bucket}$ then let $\mathsf{data}^* \leftarrow \mathsf{data}_0$ and
       remove this block from bucket.
4: end for
5: if op = read then add $(\mathsf{addr}, \mathsf{data}^*, l^*)$ to the root bucket; else add
   $(\mathsf{addr}, \mathsf{data}, l^*)$ to the root bucket.
6: Call the Evict subroutine.
7: return $\mathsf{data}^*$.
```

Algorithm 2 The procedure Evict

```
1: for each level $d$ from root to the level of leaves $-1$ do
2:     $\mathsf{bucket}_0$, $\mathsf{bucket}_1$ $\leftarrow$ randomly choose 2 distinct buckets in the level $d$
       (for the root level, pick one bucket).
3:     for bucket $\in \{\mathsf{bucket}_0, \mathsf{bucket}_1\}$ do
4:         block := pop a real block from bucket if one exists; else block :=
           $(\perp, \perp, \perp)$.
5:         for each of the two children of bucket in a fixed order: scan the child
           bucket reading and writing every block. If block is real and wants to
           go to the child, write block to an empty slot in the child bucket.
6:     end for
7: end for
```

- As we perform eviction, we pay a cost for this maintenance operation and the cost is charged to each data access. Obviously, if we are willing to pay more such cost, we can pack blocks closer to the leaves, thus leaving more room in smaller levels. In this way, overflows are less likely to happen. On the other hand, we also do not want the eviction cost to be too expensive. Therefore, another tricky issue is how we can design an eviction algorithm that achieves the best of both worlds: with a small number of eviction operations, we can avoid overflow almost surely (i.e., no overflow except with negligible in N probability).

We describe a simple candidate eviction scheme, and we will give an informal analysis of the scheme later:

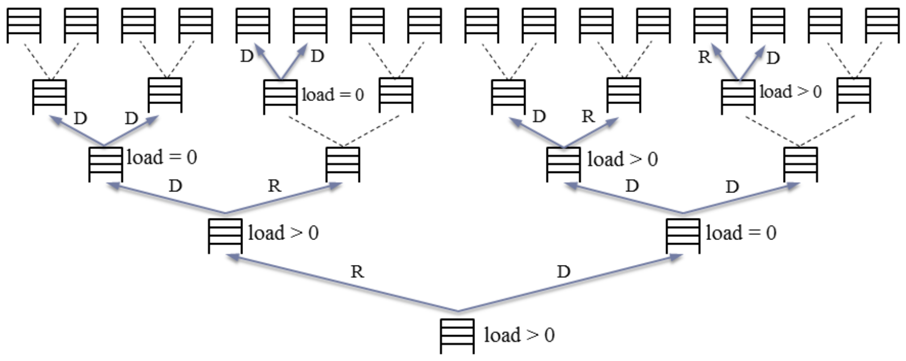

Figure 2.3: The `Evict` algorithm. Upon every data access operation, 2 buckets are chosen at every level of the tree for eviction during which one data block will be evicted to one of its children. To ensure security, a dummy eviction is performed for the child that does not receive a block; further, if the bucket chosen for eviction is empty, dummy evictions are performed on both children buckets. In this figure, R denotes a real eviction and D denotes a dummy eviction.

> *[An eviction algorithm]* Upon every data access, we choose random 2 buckets in every level of the tree for eviction (for the root level, pick one bucket). If a bucket is chosen for eviction, we will pop an arbitrary block (if one exists) from the bucket, and write the block to one of its children.

Note that depending on the chosen block's designated path, there is only one child where the block can legitimately go. We must take precautions to hide where this block is going: thus for the remaining child that does not receive a block, we can perform a "dummy" eviction. Additionally, if the bucket chosen for eviction is empty (i.e., does not contain any real blocks), then we make a dummy eviction for both children — this way we avoid leaking the information that the chosen bucket is empty.

More specifically, to write an intended block to a child bucket, we sequentially scan through the child bucket. If the slot is occupied with a real block, we simply write the block back. If the slot is empty, we write the intended block into that slot. A dummy eviction therefore is basically reading every block sequentially and writing the original contents back.

So far, we have not argued why any bucket that receives a block always has space for this block, i.e., there is no overflow except with negligible probability — we will give an informal analysis later to show that this is indeed the case (had it not been the case, the ORAM algorithm's correctness will be affected).

Algorithm pseudo-code. We present the algorithm's pseudo-code in Algorithms 1 and 2.

Remark 2.4. Note that in a full-fledged implementation, all blocks are typically encrypted to hide the contents of the block. Whenever reading and writing back a block, the block must be re-encrypted prior to being written back. If the encryption scheme is secure, the server should not be able to tell whether the block's content has changed upon seeing the new ciphertext.

Binary-Tree ORAM: Analysis

We will now discuss why the aforementioned binary-tree ORAM construction 1) preserves obliviousness; and 2) is correct except with negligible in N probability.

Obliviousness. Obliviousness is in fact easy to see. First, whenever a block is accessed, it is assigned to a new path and the choice of the new path is kept secret from the server. Thus, whenever the block is accessed again, the server simply observes a random path being accessed. Second, observe that the entire eviction process does not depend on the input requests at all.

Correctness. Correctness is somewhat more tricky to argue. As mentioned earlier, to argue correctness, we must argue why no overflow will ever occur except with negligible probability — as long as the bucket size Z is set appropriately.

Claim 2.5 (Bucket size and overflow probability)**.** *If the bucket size Z is super-logarithmic in N, then over any polynomially many accesses, no bucket overflows except with negligible in N probability.*

Proof. Note that for the leaf nodes, we can apply a standard balls-and-bins analysis, that is, if we throw N balls into N bins at random, then by Chernoff bound, we have that for any super-constant function $\alpha(\cdot)$,

$$\Pr[\text{max bin load} > \alpha \log N] \leq \exp(-\Omega(N))$$

Henceforth we focus on analyzing non-leaf buckets. We shall give a "cheating" proof, which is almost correct but to formalize it requires some extra work as explained later.

- First, observe that the root bucket (i.e., level 0 of the ORAM tree) receives exactly 1 incoming block with every access, but we get to evict the root bucket twice upon every access, and thus whatever enters the root gets evicted immediately. The root bucket is a special case and henceforth we no longer need to consider the root bucket in the analysis below.

- Now consider a bucket at level 1 of the ORAM tree. On average, one out of every two accesses (think about why), a block will enqueue in the bucket. With probability 1, the bucket will be chosen for eviction. If the bucket is chosen for eviction, a block gets to dequeue from this bucket.

- Similarly, now consider a bucket at level 2 of the ORAM tree. On average, one out of every four accesses (think about why), a block will enqueue in the bucket. With probability $\frac{1}{2}$, the bucket will be chosen for eviction.

- In general, we can conclude that for any non-leaf level $i > 1$ of the ORAM tree, with each access, one out of every 2^i accesses, a block will enqueue, and with probability $\frac{1}{2^{i-1}}$, the bucket is chosen for eviction.

Now we see a useful pattern: for every non-leaf and non-root level of the tree, with every ORAM access, the dequeue probability is twice as large as the enqueue probability. This reminds us of the standard discrete-time M/M/1 queue which you might have learned about in a basic probability class. Recall that in general, in a discrete-time M/M/1 queue,

- Every time step, with probability p, an item enqueues;

- Every time step, with probability $2p$, an item dequeues if the queue is non-empty.

Standard Markov chain analysis tells us that at any given time (prove this on your own, or alternatively we can do this proof together in a guided fashion in our homework)

$$\Pr[\text{M/M/1 queue length} > R] \leq \exp(\Omega(-R))$$

Thus, if each bucket indeed behaves like an M/M/1 queue, we could just apply this standard M/M/1 queue analysis to prove Claim 2.5 (please do the remaining work yourself: remember, it involves applying a union bound over all time steps).

Unfortunately, we cheated here. Can you spot why?

The reason is that the buckets in the ORAM tree are not independent, and our informal argument above ignored possible dependence between buckets. Well, fortunately, it turns out that this is not a big issue, and if we simply apply the discrete version of Burkes' theorem for tandem queues, we can in fact turn the above informal analysis into a formal proof! Imprecisely speaking, Burkes' theorem says that in such a tandem queuing system as the above, even though the queue lengths are not independent, it turns out that the stationary distribution of each queue's length is the same as having independent M/M/1 queues. ■

Binary-Tree ORAM: Recursion

Recall that so far, we have cheated and pretended that the client can store a large position map. We now describe how to get rid of this position map. The idea is simple: instead of storing the position map on the client side, we simply store it in a smaller ORAM denoted $\mathsf{posORAM}_1$ on the server side. The position map of $\mathsf{posORAM}_1$ will then be stored in an even smaller ORAM denoted $\mathsf{posORAM}_2$ on the server, and so on. As long as the block size is at least $\Omega(\log N)$ bits, every time we recurse, the ORAM's size reduces by a constant factor; and thus $O(\log N)$ levels of recursion would suffice.

We can thus conclude with the following theorem.

Theorem 2.6 (Binary-tree ORAM [SCSL11]). *For any super-constant function $\alpha(\cdot)$, there is an ORAM scheme that achieves $O(\alpha \log^3 N)$ cost for each access, i.e., each logical request will translate to $O(\alpha \log^3 N)$ physical accesses; and moreover, the client is required to store only $O(1)$ number of blocks.*

Note that in the total cost $O(\alpha \log^3 N)$, an $\alpha \log N$ factor comes from the bucket size; another $\log N$ factor comes from the total height of the tree; and the remaining $\log N$ factors comes from the recursion.

Epilogue

In your homework, you will implement Path ORAM [SvDS$^+$13], another very simple ORAM scheme that follows the tree-based paradigm. In comparison with the aforementioned binary-tree ORAM scheme, Path ORAM's main idea is to design a more aggressive eviction algorithm, such that the bucket size Z may be as small as $O(1)$, thus reducing the ORAM's overhead by a logarithmic factor.

On the other hand, Path ORAM requires a more sophisticated proof — thus in our lecture we chose to go over the binary-tree ORAM which admits a very intuitive "cheating" proof (which can, in fact, be formalized using just a little bit of extra work).

Acknowledgments. Thanks to Junru Shao and CS6832 students for making an initial scribe.