Privacy Enhancing Technologies
Lecture 6 – Oblivious RAM

Florian Tramèr

AGENDA

1. Recap on PIR
2. Motivating ORAM
3. Defining ORAM
4. A $\sqrt{n}$ construction
5. Modern tree-based ORAM

1 Recap

In the last lecture, we talked about Private Information Retrieval (PIR), a way for a client to privately read from a (public) database. What if we wanted to go a step further, and allow the client to privately read and write to a database? This is the idea behind Oblivious RAM (ORAM), which we will now discuss.

2 Motivation

Private Dropbox. Suppose you want to build a private version of Dropbox, where the user can upload files and download them without revealing "anything" to the server. You could of course just encrypt all the files so the server doesn't know their content. But the server can still see *access patterns* to files. This could leak a lot of information to the server (e.g., which files are accessed frequently, or at regular intervals, or in a correlated manner, etc.).

Hardware enclaves. You may have heard of hardware enclaves such as Intel SGX (if not, take Prof. Shinde's class!). These are chips that allow you to run code in an encrypted region of memory (the "enclave") that cannot be accessed by the operating system. A popular application of enclaves is for secure remote computation: you send your data to an enclave hosted in the cloud and run your application without leaking any data to the server.

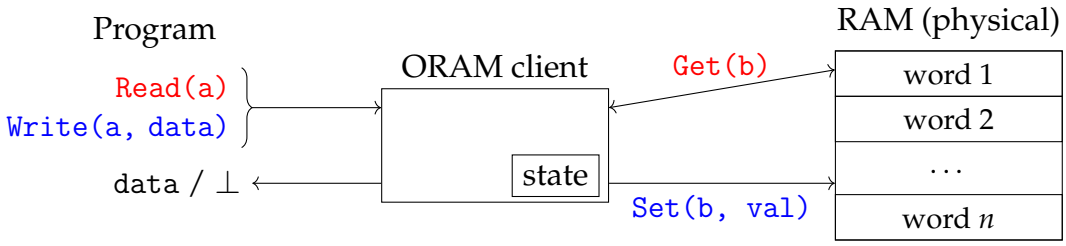
Yet here also, even though the memory is encrypted the server could still learn lots of information by observing memory access patterns, e.g.,

```
if (secret) {
    read mem[0x0]
} else {
    read mem[0x1]
}
```

Even if the memory at addresses `0x0` and `0x1` is encrypted and `secret` is stored in encrypted RAM, the server still learns the value of `secret` by observing which memory is accessed [SCNS16].

3 Defining ORAM

An Oblivious RAM [GO96] aims to address exactly the above challenges. It is a wrapper around a traditional RAM that converts a series of *logical* read and write requests from the application into *physical* reads and writes against the underlying RAM.



The properties we would like to achieve are:

- **Correctness:** The program behaves correctly as if it had access to a traditional RAM.
- **Security:** The access patterns to the physical RAM should not leak anything about the reads and writes made by the program.

Let's formalize this. For some operation $\texttt{op} \in \{\texttt{Read}(\cdot), \texttt{Write}(\cdot,\cdot)\}$, let $\texttt{Access}(\texttt{op})$ be the set of accesses made to the physical RAM when servicing $\texttt{op}$.

ORAM properties:

- *Correctness:* For any sequence $\mathcal{O} = \{\texttt{op}_1, \texttt{op}_2, \ldots, \texttt{op}_k\}$ of read/write operations from the program, the ORAM client (when talking to an honest RAM) answers each operation correctly.
- *Security:* Let $\mathcal{O} = \{\texttt{op}_1, \texttt{op}_2, \ldots, \texttt{op}_k\}$ and $\mathcal{O}' = \{\texttt{op}'_1, \texttt{op}'_2, \ldots, \texttt{op}'_k\}$ be any two poly-size sequences of equal length. Then it holds that

  $$\{\texttt{Access}(\texttt{op}_1), \ldots, \texttt{Access}(\texttt{op}_k)\} \overset{c}{\approx} \{\texttt{Access}(\texttt{op}'_1), \ldots, \texttt{Access}(\texttt{op}'_k)\}$$

  That is, the ORAM leaks nothing except the *number* of memory accesses made by the program.

Trivial solutions.

- *No outsourcing:* The ORAM client stores the entire n-word RAM in its internal state (the "stash") and never accesses the physical RAM.
  - n words of storage.
  - 0 RAM accesses per op.
- *Linear scans:* On every op, the ORAM client reads the entire physical RAM (and freshly re-encrypts each word).
  - $O(1)$ words of storage (an encryption key).
  - n RAM accesses per op.

Efficiency goal. We of course want an ORAM that is more efficient than these trivial solutions. Concretely, we want an ORAM with small storage and few RAM accesses per op.

It is known that in an "online" setting (where operations arrive sequentially), the best we can hope to achieve is $O(\log n)$ RAM accesses per op, even if the ORAM stores as much as n^ε words (for constant $\varepsilon > 0$) [LN18].

There are schemes that achieve this bound, and are thus tight [AKL+20]. The schemes we present here have slightly worse asymptotic overheads, but are much more practical.

Comparison with PIR. Informally PIR is a way to privately *read* from a (public) database, while ORAM is a way for a client to privately *read and write* to a (private) RAM. So one may be tempted to think that PIR is a special case and simpler version of ORAM. But this is not the case, as the setups differ in important ways:

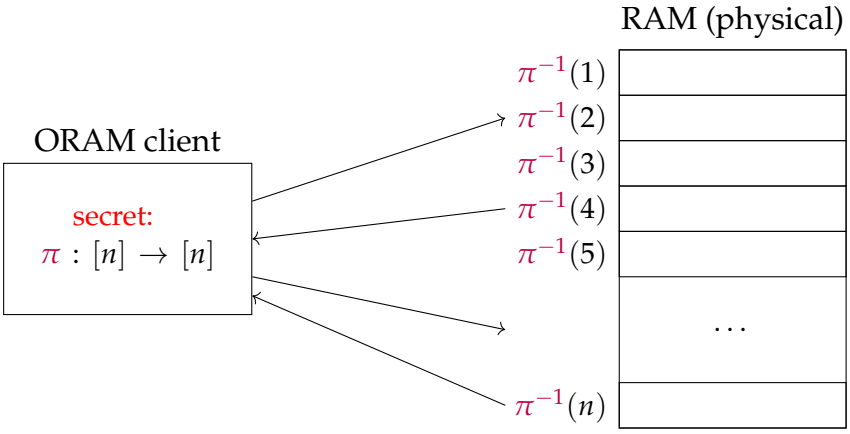
PIR	**ORAM**
"Private reads from public database"	*"Private reads & writes to private RAM"*
Database is public and static	Server memory is private and changes on every op
Many clients ⇔ one server	one client ⇔ one server
Server does linear work per query[a]	Server can process ops with poly-log overhead
Client is stateless[b]	Client has small private state
Requires public-key crypto in single-server setting	Can be built using just PRFs!

[a] Can be amortized with pre-processing
[b] Some pre-processing schemes require keeping state

4 A $\sqrt{n}$ Construction

We'll start with a very nice and simple ORAM construction with an $O(\sqrt{n})$ overhead, inspired by the original construction of [GO96].

The idea behind this scheme is quite simple: suppose the contents in the RAM were permuted according to some random permutation π that is only known to the ORAM client. Then, any sequence of $k \leq n$ RAM operations that access *distinct* locations are indistinguishable (regardless of whether the program needs to do a read or a write at address a, the ORAM client will read address $\pi(a)$, and rewrite it with a fresh re-encryption of the memory contents).



Of course, in practice a program might need to access the same location multiple times. So what we will do is *re-shuffle* the RAM contents periodically to make sure that the ORAM client never has to access the same location in RAM in-between reshuffles.

Oblivious sorting. To begin, we'll discuss oblivious sorting algorithms that have many applications beyond ORAM.[1] These are algorithms that sort memory with a RAM access pattern that is *independent* of the input data. These are also called *sorting networks*, as they can be implemented by a fixed network of wires and comparators.

One (bad) example you might be familiar with is Bubble sort: the algorithm iterates over all pairs of elements in the array, and swaps them if they are in the wrong order. After the first iteration, the largest element is now the last element in the array. We then repeat the process on the first $n-1$ elements, and so on. This requires $O(n^2)$ oblivious RAM accesses.

A better alternative is Batcher sort [Bat68], which has a complexity of $O(n \log^2 n)$. There are more complicated algorithms which require only $O(n \log n)$ oblivious RAM accesses (i.e., the same complexity as for non-oblivious algorithms), but they are unlikely to be used in practice.

The full protocol. The protocol now proceeds as follows:

`init:`

- The client ORAM keeps a stash of $\sqrt{n}$ words in its internal state, initialized to zero.
- The RAM is initialized to encryptions of zero.

`while(true):`

- The client samples a fresh random permutation π (e.g., using a PRG) and stores the key for π in its internal state.
- The client shuffles the entire n-word RAM according to π. If the stash is not empty, the client stores it back in the RAM.
- Process $\sqrt{n}$ ops:
  - If the address is in the stash, access a random address in the RAM and discard the result.
  - Otherwise, read the address in the RAM and store it in the stash.
  - If the operation is a write, perform it on the copy from the stash.

The ORAM client does an oblivious shuffle which takes $O(n \log^2 n)$ time, and then processes $\sqrt{n}$ ops which each require a single RAM access. Thus, the amortized cost per access is $\tilde{O}(\sqrt{n})$.

As described, the protocol requires the client to keep a stash of size $O(\sqrt{n})$ in its internal state. We can also store the stash in the RAM itself and read it back in before each iteration, without affecting the asymptotic complexity.

[1] In ORAM we want to *shuffle* the data, not sort it. But the same algorithms can be used for both.

5 Tree-based ORAM

A $\sqrt{n}$ overhead is still pretty large. Worse, the overhead is not uniformly distributed so every once in a while your program just halts for a long time while it reshuffles the entire RAM.

Modern schemes fix these issues. Tree-based ORAM schemes were developed in a series of works starting with [SCSL11, SDS+18]. These have a much smaller overhead of $O(\text{polylog } n)$, and the overhead is uniformly distributed across all operations. The scheme we will cover, called "Simple ORAM" [CP13], is not the most efficient scheme to date (in theory or practice), but it illustrates the main ideas behind tree-based constructions, and has the advantage of being super simple to explain!

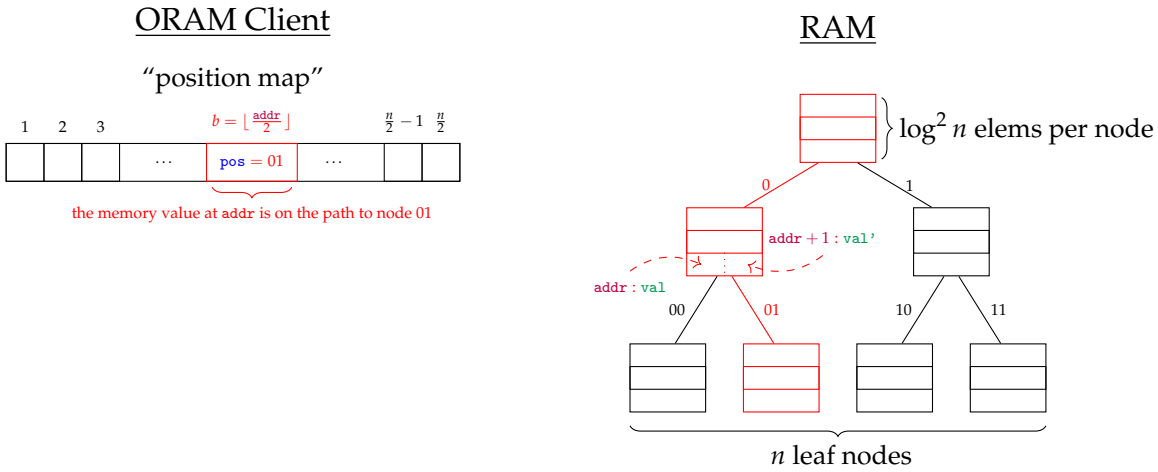
This scheme is based on the following idea:

1. We start by building a "bad" ORAM in which the client's stash is of size $n/2$ instead of n, while incurring $O(\log^3 n)$ overhead per RAM access.
2. Recurse, by storing the stash in *another ORAM* (does this remind you of recursive PIR?).
3. Repeat the recursion $\log n$ times to get a stash of constant size. The total overhead is then
$$O(\log^3 n) \cdot O(\log n) = O(\log^4 n)$$

This elegant recursive idea was introduced in [SCSL11]. We just have to explain how to build the bad ORAM in step 1.

The RAM will be stored in a binary tree of depth $\log n$, where each node can contain $\log^2 n$ memory elements. We ensure that the elements at addresses $2k$ and $2k+1$ are always stored adjacent to each other. The client's stash contains a "position map" of size $n/2$. The construction maintains the following invariant at each stage:

> The data for address `addr` is stored on the path from the root to the leaf node indicated at position $\lfloor \texttt{addr}/2 \rfloor$ in the position map.



ORAM operations. OK, so how do we read and write in this thing?

`Read(addr)` :

1. Look up the position `pos` of the leaf node at element $b := \lfloor \texttt{addr}/2 \rfloor$ in the position map.
2. Read contents of all nodes on the path from the root to the leaf node `pos`, to find the data at address `addr` and $\texttt{addr} + 1$ (assume `addr` is even).
3. Pick a new random leaf node `pos'` $\leftarrow^{\$} [n]$ and update $\texttt{PosMap}[b] \leftarrow \texttt{pos'}$.
4. Encrypt (`addr`, `val`) and (`addr+1`, `val'`) and add them to the root node.[a]
   If the root node is full, output: OVERFLOW.
5. Pick a random leaf node $l \leftarrow^{\$} [n]$. Walk down the path from the root to node l, and *flush* all elements as far down as possible, while maintaining the invariant. That is, all elements are pushed to the lowest common ancestor of node l and the flushed element's leaf node. (for example, if `pos'` $= 00$ and $l = 01$, we would flush the newly inserted data down to node 0).
   If any node is out of space, output: OVERFLOW.

[a] For a write, just update `val` beforehand.

Correctness: As long as no overflow occurs, all read/write operations return the right values.

Security: For every operation, the client reads & writes all data along two uniformly random paths in the tree, and so the server learns nothing about the accessed position!

Overhead:

- Client storage: $n/2$
- Read/write overhead: $O(\log n)$ nodes of $\log^2 n$ elements each, so $O(\log^3 n)$.
- Server storage: stores $O(n)$ nodes, so $O(n \log^2 n)$ total.

Bounding overflow:

- Bounding overflow at the leaves: Each element is assigned a leaf at random. By a Chernoff bound, we can show that any leaf overflows with negligible probability.
- Bounding overflow at internal nodes: This analysis is slightly more technical, and involves showing that an overflow requires a large number of memory elements being assigned to some leaf node $\texttt{pos}'$, without ever flushing the path to this leaf. Since leaf assignments and flushing leaves are both chosen at random, we can bound this probability to be negligible (see [CP13] for details).

References

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: optimal oblivious RAM. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Appli-*

cations of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30, pages 403–432. Springer, 2020.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. *Cryptology ePrint Archive*, 2013.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328, 2016.

[SCSL11] Elaine Shi, T H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*, pages 197–214. Springer, 2011.

[SDS+18] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.