Oblivious RAM

Notes by Henry Corrigan-Gibbs

MIT - 6.5610
Lecture 20 (April 23, 2025)

> ***Warning:*** This document is a rough draft, so it may contain bugs. Please feel free to email me with corrections.

Outline

- Motivation
- Formal definition

Introduction

Our topic today, by popular demand, is *oblivious RAM* (ORAM). Oblivious RAM is one of these cryptographic tools that feels like it *should* have loads of applications, but has yet to make its way into wide use.

The notion of ORAM is due to Goldreich and Ostrovsky [2].

Informally, an ORAM scheme lets a client outsource a set of N data items to a server while maintaining:

1. *Data privacy:* The server learns nothing (in a semantic-security sense) about the data, and
2. *Obliviousness:* The client can read or write/update the data items without revealing to the server which items it is accessing.

We will formalize these properties in a moment.

Before we do, notice that the first feature is easy to achieve with any symmetric-key encryption scheme. The second feature is the one that requires some work.

Applications

Before diving into what ORAM is, let me sketch a few scenarios in which we might want to use it.

Outsourced file storage. You want to back up all of the files on your hard drive to Dropbox. You would like to read and write files without Dropbox learning which files you are reading and writing.

You are the ORAM client; Dropbox is the ORAM server.

Physical attacks. In certain settings, it may be relatively easy for an attacker to read the contents of your computer's RAM (e.g., by snooping on the memory bus) but it might be difficult for the attacker to look inside the CPU. If you run all of your programs using an ORAM scheme, an attacker controlling the RAM will not learn anything about your data or data-access pattern.

Your CPU is the ORAM client; the RAM is the ORAM server.

RAM multi-party computation. There is also a nice way to use oblivious RAM to get asymptotically efficient multiparty computation protocols for RAM programs. I will not say more about this here.

Hardware enclaves. Some modern CPUs support *hardware enclaves*: an environment that allows running a user-space process while preventing even kernel code from being able to read the memory of the process in the enclave. If a process running in an enclave wants to persist data to a disk, it could use an ORAM scheme to both hide the data contents and its access pattern.

Signal apparently uses ORAM blog post for this application in its contact-discovery system.

Why is it important to hide access patterns?

You might think that encrypting user data is enough—why do we need to hide data-access patterns too? In many applications, the memory-accesses themselves leak your data. Signal's contact-discovery application is an example.

Definition: Oblivious RAM

We use definitions loosely following Asharov et al. [1].

A RAM machine with memory size n consists of a CPU, a small number of registers, and a size-n external memory. The registers and memory entries each hold a w-bit "word," where we always assume the word size $w \geq \log n$.

Let P be a RAM program (e.g., an x86 program). An *oblivious simulation* of P is a program $\widetilde{P}$ that preserves the functionality of P, but whose memory-access pattern "leaks nothing" about memory accesses that P makes. We can make this notion formal using simulation.

To do so, for a RAM program P, let $\mathsf{addrs}(P)$ denote the sequence of memory operations that P makes during its execution. Each operation is either a read $(\mathtt{R}, \mathsf{addr})$ or a write $(\mathtt{W}, \mathsf{addr}, \mathsf{value})$.

Elaine Shi has a nice set of notes on oblivious RAM.

At the last minute, I decided to simplify the ORAM definition here. When I attempt simplification, I end up going to far 50% of the time and breaking something by mistake. So I hope that I did not do that here. But at least even if it is broken, it is simple. :)

We will also assume that the word size is greater than the security parameter λ.

Definition 1 (Oblivious RAM)**.** We say that a p.p.t. algorithm $\mathcal{O}$ is an *oblivious RAM scheme* if it satisfies the following properties:

- *Correctness.* For all $\lambda \in N$ and all ram programs P, for $\widetilde{P} \leftarrow \mathcal{O}(1^\lambda, P)$ it holds that $P() = \widetilde{P}()$.
- *Obliviousness.* There exists a p.p.t. simulator Sim such that for all programs P, the following probability distributions ensembles are computationally indistinguishable:

$$\{\mathsf{addrs}(\mathcal{O}(1^\lambda, P))\}_{\lambda=1}^{\infty} \overset{c}{\approx} \{\mathsf{Sim}(1^\lambda, 1^{|\mathsf{addrs}(P)|})\}_{\lambda=1}^{\infty}.$$

To unpack the second part here: we are requiring that there exists an efficient simulator that can produces a transcript of the memory accesses that $P(x)$ makes. The simulator does *not* take the program P as input, only the number of memory accesses that P makes. This implies that the memory-access pattern of $\mathcal{O}(1^\lambda, P)$ reveals essentially no information about the memory accesses that the original program P makes.

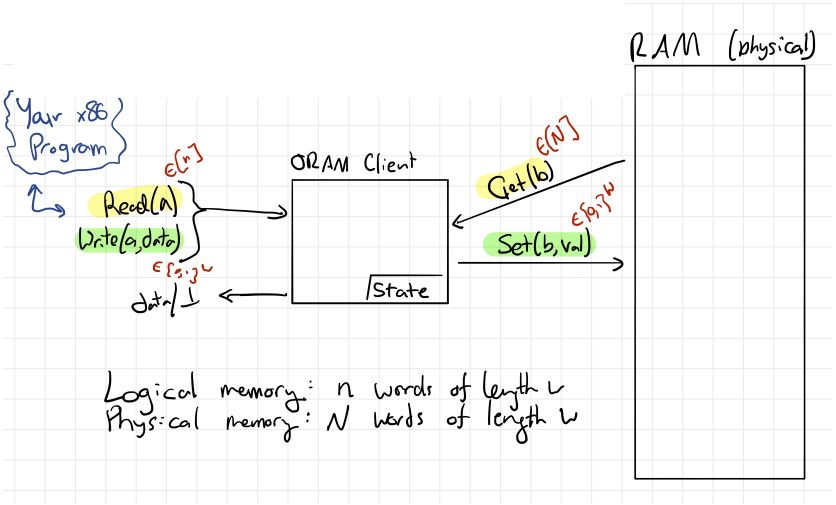


Figure 1: Oblivious RAM

Important caveat: Oblivious RAM—at least as we have defined it here here—does not guarantee any kind of memory integrity. Neither component of this definition gives any guarantee when the program is executed against a RAM that does not behave as an honest RAM would.

Many applications of ORAM demand integrity as well; it is possible to boost ORAM to also provide integrity protection using Merkle trees. If you want to do so with very minimal overhead, you have to work harder [4].

The following two notions of efficiency are the most important ones for us:

Definition 2 (ORAM efficiency metrics)**.** Let $\mathcal{O}$ be an efficient algorithm such that for all RAM programs P using at most n words of memory, $\widetilde{P} = \mathcal{O}(P)$ obliviously simulates P. We say that $\mathcal{O}$ is an *oblivious RAM scheme* with :

- *memory overhead* $m(n)$ if $\mathcal{O}(P)$ uses $m(n) \cdot n$ words of memory,
- *computational overhead* $c(n)$ if, $\mathcal{O}(P)$ makes $c(n) \cdot |\mathsf{addrs}(P)|$ memory accesses.

Simplifying assumption: The data is encrypted. When we discuss ORAM schemes from here on, we implicitly assume that the scheme encrypts each stored word of memory using a symmetric-encryption scheme, whose key is stored in a register. This only uses one extra register and does not change the number of memory accesses needed. With this assumption, we only need to worry about access-pattern leakage—not data leakage.

Oblivious RAM versus Private Information Retrieval (PIR)

Both oblivious RAM and PIR involve hiding a client's access patterns from a potential adversarial server. It is important to understand how they differ.

ORAM	**PIR**
Server's memory changes with each query	DB is public, static
One client $\Leftrightarrow$ one server	Many clients talk to one server
Supports reads and writes	Supports only private reads
$\mathsf{polylog}(n)$ time per op (memory size n)	$\Omega(n)$ time per op (database size n)
Can build from PRFs	Requires public-key crypto in single-server setting

Sanity checks: Trivial solutions

CPU with a gigantic number of registers. Since reads and writes to registers are not visible to the RAM, if the CPU has n registers, it is possible to obliviously simulate any program that uses at most n memory words with zero memory and compute overhead: just store all of the data in registers!

Normally, we think of the number of registers as being constant or maybe polylogarithmic in the memory size, which rules out this trivial solution.

Read all of memory to implement each access.

Claim. There is an oblivious RAM scheme with:

- memory overhead $O(1)$ and
- computational overhead $O(n)$.

Proof sketch. The idea is to run the program P as usual except that, whenever P asks to read/write location $i^* \in [n]$ in memory, read all n words of memory, one at a time, and write each back in place. When we reach the index i^* we care about, we can modify the contents or store it to a register.

Correctness is immediate.

For security, notice that the memory access pattern of $\mathcal{O}(P)$ depends only on the number of memory accesses that P makes. □

This trivial scheme is terrible in terms of performance, but it at least shows us that oblivious simulation is possible!

Goal: Few registers and little overhead

How good can an ORAM scheme be?

- Larsen and Nielsen (2018) [3] showed that even with n^ϵ registers, the total CPU-RAM communication must increase by a $\Omega(\log n)$ factor on memory size n.
- Path ORAM (2012) is a simple scheme that achieves this bound when the word size is $\Omega(\log^2 n)$ [5].
- OptORAMa [1] (2020) is a complicated scheme that gets rid of the word-size restriction.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: optimal oblivious RAM. In *EUROCRYPT*, 2020.

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

[3] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, 2018.

[4] Surya Mathialagan and Neekon Vafa. MacORAMa: Optimal oblivious RAM with integrity. In *CRYPTO*, 2023.

[5] 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.

The "Square-Root ORAM" (Goldreich & Ostrovsky '92)

Simple + clean. We will see a more efficient construction next class.

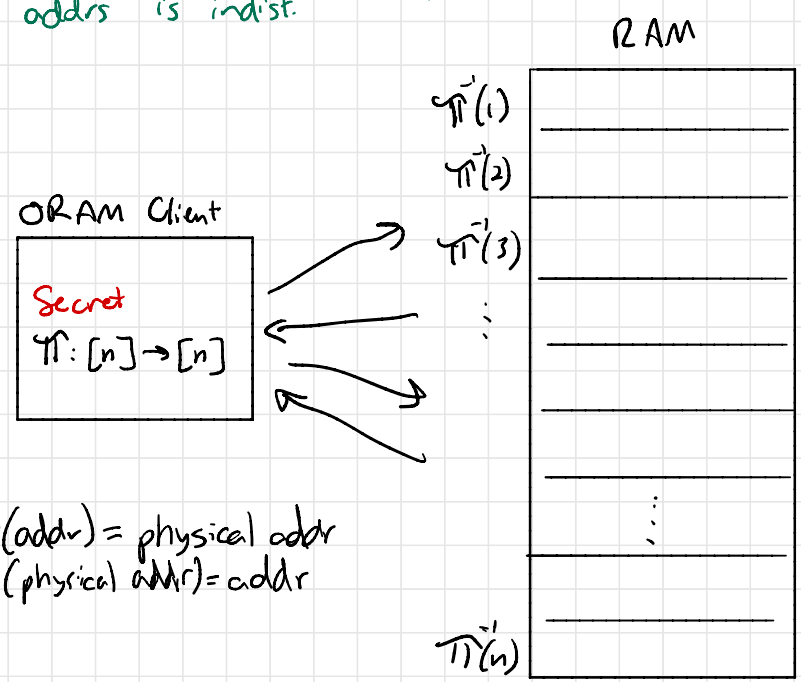
Client storage: $O(1)$ words (PRF key)

Server storage: $n + O(\sqrt{n})$ words

$O(\sqrt{n})$ RAM accesses per op., amortized

Key idea: Suppose RAM holds logical mem contents permuted according to some π that only client knows.

$\Rightarrow$ "Read-once ORAM"... any sequence of ops to distinct addrs is indist.



$\pi(\text{addr}) = \text{physical addr}$

$\pi^{-1}(\text{physical addr}) = \text{addr}$

Construction

* Initialize memory contents with encryption of 0s (using sem sec enc scheme)
  n data blocks, $\sqrt{n}$ dummy blocks, $\sqrt{n}$ stash blocks

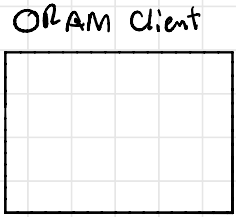
* While true:

  1. Shuffle $n+\sqrt{n}$ data + dummy blocks using fresh random perm $\pi: [n+\sqrt{n}] \to [n+\sqrt{n}]$.

  2. Process $\sqrt{n}$ ops:

     - Read + write back entire stash
     - If desired element is in stash
       → read one dummy block
     - Else
       → read data block
     - Read + write back entire stash

  3. Return all words to their starting location.



Details

Step 1: Sorting.

* Use PRP to assign tag $\pi(i)$ to addr i.
* Run a sorting network to sort by tags in $O(n \log^2 n)$ RAM access

Step 2: Access

* Read stash: $\sqrt{n}$ RAM accesses
* Read data/dummy elem: 1 "
* Read stash: $\sqrt{n}$ "

Step 3: Unsort, again using Batcher

$O(n \log^2 n)$ RAM accesses

Total cost: $O(n \log^2 n)$ RAM accesses per $\sqrt{n}$ logical ops

$\Rightarrow$ Amortized $O(\sqrt{n} \log^2 n)$ cost per access.

So, we saw that ORAM is possible w/ sym-key tools w/ $O(\sqrt{n}\log^2 n)$ overhead per access.

Next time: a more efficient scheme.

Tree-Based ORAM

Developed in a long series of really nice papers. Relatively recent
↳ Shi, Chan, Stefanov, Li (2011) + work after

We will see "Simple ORAM" of Chung & Pass (2013)

Client storage: $O(\log^2 n)$

RAM storage: $O(n \log^2 n)$

Comp overhead: $O(\log^4 n)$ R/W to RAM per logical op.

Remember: More recent ORAMs give improvements in theory & practice. But this is simple.

Plan: 1. Construct a "bad" ORAM in which client stores $n/2$ blocks instead of n.
↳ $O(\log^3 n)$ comp overhead

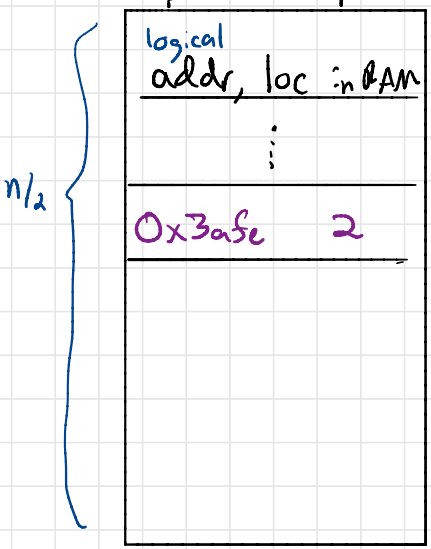
2. Recursively store the $n/2$ blocks in another ORAM... recurse all the way down

Overhead: $O(\log^3 n) \cdot \log n$
b/c of recursion.

Only need to explain Step 1.

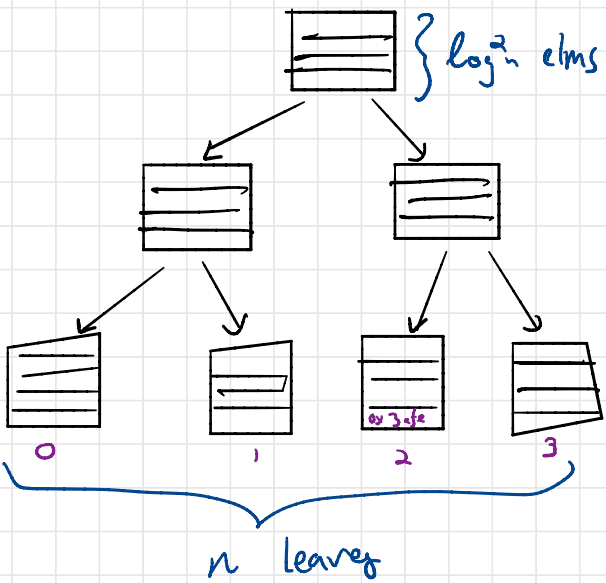
ORAM Client

"position map"



(Store pairs of logical addrs next to each other)

RAM



Invariant: data for addr is stored on path to leaf indicated in position map.

ORAM Operations

- Read & Write are essentially the same.
- Let's look at a read...

Read(addr)

1. Look up leaf ℓ of addr in position map.
2. Read contents of all buckets on path to leaf ℓ.
3. Pick a new random leaf $\ell' \xleftarrow{R} [\cdot]$, PosMap[addr] $\leftarrow \ell'$.
4. Add (addr, data) ... encrypted to root bucket.
   - ↳ If no space, fail.
5. Pick a leaf $\ell \xleftarrow{R} [n]$, walk down path from root to ℓ.
   - ↳ "flush" blocks down towards ℓ as far as they can go while still maintaining the Invariant.
   - ↳ If no space, fail.

That's it! So simple!

For writes, just update new contents before putting data back into tree root.

Properties

Correctness: As long as there's no overflow, all read/write ops return right answer. ✓

Security: On each R/W, client reads two random paths from root to leaves. ✓

Overhead:
- $n/2$ client storage
- Buckets have size $\log^2 n$, need to read/write $O(\log n)$ of them $\Rightarrow O(\log^3 n)$.
- Server stores $O(n)$ buckets $\Rightarrow O(n \log^2 n)$.

To show there's no overflow.
- Bound leaf overflow: Chernoff bound.
- Bound node overflow: Slightly more involved, but still not too bad.
  ↳ See Pass paper.

Summarizing

- ORAM lets a client outsource its storage while hiding access patterns.
- Best constructions have logarithmic overhead (in # of ram accesses) & have varying levels of practicality.
  - Not sure whether any deployed systems have used ORAMs...
    Can speculate on why not.