

Oblivious RAM

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2026

So far, we have considered secure protocols that handle programs expressed as Boolean circuits. As we will investigate today, while Boolean circuits are theoretically sufficient to encode any computable function, that encoding can be quite expensive in terms of the number of required gates. This is problematic, since in all protocols we have considered, costs scale with circuit size.

The main problem that we will begin to investigate today is that of securely instantiating random access memory (RAM). Namely, suppose our parties wish to run a program that repeatedly reads/writes some large memory. This is, of course, an understandable wish — random access memory is the central ingredient in a huge variety of algorithms. Our end goal will be to develop methods by which we can securely and efficiently instantiate memory in the context of an MPC protocol.

The tool by which we will achieve such protocols is a powerful cryptographic technique called *oblivious RAM* [GO96].

Some Vocabulary; Word Size

We will consider arrays M of length n . Each entry M_i is itself a bitstring that we will refer to as a *word*. Each word has some specified length w . It will be helpful to assume that w is at least logarithmic in n ; this is because $\log n$ bits are needed simply to index the array M itself. To build useful algorithms it is useful to be able to store array indexes inside array cells, allowing to build complex data structures. So, we will assume that $w = \Omega(\log n)$. In addition, we typically wish to enforce that the word size is *no larger* than is required, as all costs will ultimately scale with w . In particular, we will assume that the word size is *proportional* to the logarithm of n , namely that $w = \Theta(\log n)$.

Why is RAM expensive in the context of a Boolean circuit?

To start, let us understand why there is a problem with the techniques that we have developed so far. Namely, let us understand why it is inefficient to implement random access arrays with Boolean gates. Note that if we could do so efficiently, then we could indeed instantiate arrays inside MPC protocols, by simply compiling RAM accesses to corresponding Boolean circuitry.

As an attempt, let us assume that we have $n \times w$ Boolean wires that jointly encode the array M , and let us assume that we have $\log n$ wires that encode an index i . Our goal is to construct a Boolean circuit that reads out M_i .

Unfortunately, straightforward reasoning shows that we will need a circuit of size at least $\Omega(nw)$ gates to correctly and in general read out M_i . Indeed, it is necessary for our Boolean circuit to at least “read” each of the bits of the memory, of which there are nw . The manner by which we read a wire is by inserting some gate. (If there is some bit of the memory that the circuit *does not* read, then we can construct an input for which our circuit will simply have to “guess” the output, which it cannot do reliably.)

On the positive side, it is straightforward to construct a circuit of size $O(nw)$ that indeed correctly reads. This circuit is called a *linear scan*. In a linear scan we compute from i a length- n *one-hot* encoding $\text{one-hot}(i)$, which is a length- n bitstring where all entries are 0, except the i -th entry, which holds 1.

$$\text{one-hot}(i)_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Exercise 1. *It is easy to describe a Boolean circuit of size $O(n \log n)$ that computes from i the one-hot encoding $\text{one-hot}(i)$, but it is also possible to do better. Construct a Boolean circuit computing one-hot that has size $O(n)$.*

We can then use Boolean gates to compute the sum of entrywise products between array cells and entries of the one-hot string:

$$\left(\bigoplus_j M_j \cdot \text{one-hot}(i)_j \right) = M_i$$

This circuit has $O(nw)$ gates.

As discussed above, if we wish to perform only a single access to memory, then we cannot hope to do better than using a circuit of size $\Theta(nw)$. The real problem with Boolean-circuit-based RAM is when we consider what happens when we read from memory *more than once*. In particular, suppose we read memory T times. It is not clear how to construct a standard Boolean circuit that correctly solves the problem except by simply instantiating T copies of a linear scan, consuming $\Theta(Tnw) = \Theta(Tn \log n)$ gates. Here, it is important to remember that in standard Boolean circuits, we cannot “re-use” our gates in the same way that, e.g., our hardware computers re-use their gates over time.

So, if we wish to construct an MPC protocol that supports random access memory, then a Boolean-circuit-based solution will incur cost scaling with $O(nwT)$. Contrast this with your experience in cleartext computing, where we commonly assume that each RAM access incurs *constant* cost, and hence an algorithm that reads/writes memory T times will incur cost $\Theta(T)$. This is a significant penalty.

As we will see, it is also an *unnecessary* penalty. Indeed, we will study a technique called *oblivious RAM*. In the context of this course, we will see we can reduce cost from $O(Tn \log n)$ to only $O(T \log^3 n)$, a *dramatic* improvement.

The Client-Server Setting

To get a grasp on how to achieve improved cost of RAM, we will for now move away from the general MPC setting, and instead focus on a considerably simpler setting called the client-server setting.

Consider an interaction between two parties: a *client* and a *server*. Suppose the client is memory constrained, but the server is not. The client therefore wishes to *outsource* its memory storage requirements to this server. Namely, the server stores the array M ; when the client wishes to write some value x to array slot i , she simply sends (write, i, x) , and the server obliges by setting $M_i = x$. Similarly, when the client wishes to read some array slot, she simply sends (read, i) to the server, and the server responds with M_i . However, let us suppose that the client does not fully trust the server. In particular, we wish to provide security against a semi-honest server.

As a first step, the client can simply *encrypt* all data before storing it on the server. This prevents the server from reading the client’s stored data, but the truly difficult problem remains: the server still observes the *order* in which the client reads the memory addresses. It turns out that this problem is quite insidious. The goal of an *oblivious RAM* is to *obfuscate* the client’s *memory access pattern*. Since encryption handles the contents, from now on we fold it into our security model and track only *which* slots are accessed.

Oblivious RAM

In the client-server setting, it is instructive to think of the client C as a memory-accessing program that takes as input some string x , and to think of the server S as a simple memory of size n that responds to those requests in the obvious way. Namely, we consider the interaction $C \leftrightarrow S$. The goal when studying Oblivious RAM is, again, to hide the client’s memory usage from the server. In particular, we wish to instantiate the following ideal functionality:

Definition 1 (ORAM Functionality). *The functionality $\mathcal{F}_{\text{ORAM}}$ interacts with a client C and a server S . It has parameters n and w . $\mathcal{F}_{\text{ORAM}}$ is stateful, and it maintains an array M of size n , where each memory cell is of size w . The functionality responds to two types of commands:*

- **Writes.** *The client sends an array index i and an element x . The functionality sets $M[i] := x$. It sends a message **access** to the server.*

- **Reads.** The client sends an array index i . The functionality sends $M[i]$ to the client. It sends a message **access** to the server.

Remark 1. One noteworthy point here is that this ORAM functionality is reactive: it keeps state and answers a sequence of requests chosen adaptively by the client program. Our semi-honest composition theorem extends to this setting with no change to its proof.

Notice that the only information the server observes is n , w , and the total number of memory accesses. In particular, it does not learn what is stored in memory, nor which order memory is accessed in.

As already stated, the difficult part of ORAM is not hiding memory content, but simply hiding the access pattern. We can formalize by defining yet another ideal functionality. Specifically, we will define the following unprotected memory access (UMA) functionality [GHL⁺14]:

Definition 2 (UMA Functionality). The functionality $\mathcal{F}_{\text{UMA}}$ interacts with a client C and a server S . It has parameters n and w . $\mathcal{F}_{\text{UMA}}$ is stateful, and it maintains an array M of size n , where each memory cell is of size w . The functionality responds to two types of commands:

- **Writes.** The client sends an array index i and an element x . The functionality sets $M[i] := x$ and sends (**write**, i) to the server.
- **Reads.** The client sends an array index i . The functionality sends $M[i]$ to the client and sends (**read**, i) to the server.

Notice that this ideal functionality is identical to ORAM, except that it leaks (1) which type of memory access type occurs and (2) which memory address was accessed.

Given a client with only $O(\lambda)$ bits of space, it is straightforward to instantiate $\mathcal{F}_{\text{UMA}}$ from any CPA-secure encryption scheme.

Exercise 2. Construct a protocol that securely instantiates $\mathcal{F}_{\text{UMA}}$ from any CPA-secure encryption scheme. Prove security by constructing a simulator for the server (note that simulating the client is trivial; there is nothing to hide).

Now, with UMA security defined, we can define the notion of an *ORAM compiler*, which is formally what we will construct:

Definition 3 (ORAM Compiler). An **ORAM Compiler** is a client/server protocol that securely realizes ORAM in the UMA-hybrid model, where all client/server interactions pass through the UMA functionality. The overhead of the compiler is the number of hybrid UMA accesses that happen as a consequence of each ORAM access.

Unpacking the definition: Π is correct if the client program sees the same answers it would from $\mathcal{F}_{\text{ORAM}}$, and Π is secure if the sequence of indices it sends to $\mathcal{F}_{\text{UMA}}$ can be simulated given only the number of accesses. In one line: an ORAM compiler reduces an array’s leakage from “which slot” to “an access happened.” Note that the definition says nothing about cost; overhead is a separate measure, and we will see that amortized and worst-case overhead can differ.

Trivial ORAM

Let’s start with a naive ORAM compiler that nevertheless works: Our goal is to construct a compiler Π that is both correct and secure.

The trivial ORAM is, again, based on the notion of a linear scan. Namely, when the client program C wishes to read some particular memory slot i , the compiler Π simply downloads the entire memory from S , one element at a time, writes that element back, and discards all read elements except M_i . For writes, the compiler one by one downloads each array slot, then writes back the element as it was before, except for in the case of the desired element i . Here, the compiler performs the write as intended.

Notice that this is clearly correct. It also clearly has a simulatable memory access pattern: On each access, the server observes a read and a write to each memory slot. Thus, this protocol is *perfectly* secure. On the other hand, it is also incredibly inefficient in terms of overhead.

Exercise 3. Prove this trivial compiler realizes ORAM in the UMA hybrid model.

ORAM with $\tilde{O}(\sqrt{n})$ overhead

We will construct an ORAM compiler with asymptotically-superior overhead. In particular, we will see how to achieve $\tilde{O}(\sqrt{n})$ overhead, where $\tilde{O}(\cdot)$ omits $\text{poly}(\lambda, \log n)$ factors [GO96]. The main insight behind *all* ORAMs is that while we cannot hope to improve over trivial ORAM for a *single* access, we *can* hope to improve in an *amortized sense*.

The main idea of this improved ORAM is to take time on the first access to *randomly shuffle the memory stored on the server*. Then, on subsequent accesses, the compiler can quickly look for the item while keeping the memory access hidden. Every now and then, we will “use up” the shuffle on memory, and will need to reshuffle. However, this will happen infrequently enough that we will amortize the cost of shuffles.

The protocol we will look at is built on the following lemma, which we will not prove today:

Lemma 1. *There exists a protocol in the UMA hybrid model by which a constant-space client can uniformly shuffle a subarray (stored in the UMA functionality) of size n with total overhead $\tilde{O}(n)$ [AS96].*

In addition:

Lemma 2. *There exists a protocol in the UMA hybrid model by which a constant-space client can sort a sub-array (stored in the UMA functionality) of size n with total overhead $\tilde{O}(n)$.*

Both results above are based on *sorting networks* [Bat68].

Now, here is the ORAM protocol. The client holds a PRF key k and uses PRF outputs as *tags*; we take the tag length large enough that distinct inputs collide with negligible probability. The protocol proceeds in *epochs* of $\sqrt{n}$ accesses each. At the end of each epoch, the client performs an expensive refresh step, but this cost is amortized across the epoch.

Layout. The client marks off two regions of server memory:

- *Storage:* $n + \sqrt{n}$ slots, holding the n real items and $\sqrt{n}$ dummies. Real item i is tagged $F_k(i)$; the j -th dummy is tagged $F_k(n + j)$. Each slot holds a pair (tag, word).
- *Stash:* $\sqrt{n}$ slots, initially holding $\perp$.

Epoch setup. The client samples a fresh key k , assigns tags as above, and obviously sorts storage by tag (Lemma 2). To the server, storage is now permuted by a fresh pseudorandom permutation. Because storage is sorted by tag, the client can find the slot holding tag t via binary search, reading $O(\log n)$ slots.

Access j of the epoch. Suppose the client program requests logical address i (a read, or a write of value x). To handle this, the client proceeds as follows:

1. *Scan the stash.* Read all $\sqrt{n}$ stash slots, remembering whether item i is in the stash and, if so, remembering its most recent value (the copy in the highest slot).
2. *Read storage once.* If i is not in the stash, binary-search for tag $F_k(i)$ and read that slot; this yields the current value of i . Otherwise, binary-search for tag $F_k(n + j)$, the j -th dummy, and read that slot.
3. *Append to the stash.* Write the current value of item i (replaced by x on a write) to stash slot j . Return the value to the client.

The invariant is simple: an address accessed earlier in this epoch has its latest value in the stash, and its storage slot is never touched again; every other address has its latest value in storage. Therefore, if no two tags collide, the protocol is perfectly correct. By setting the size of tags high enough, collisions occur with negligible probability.

Epoch end. After $\sqrt{n}$ accesses, the stash is full and every dummy has been used, so the client rebuilds storage. It concatenates stash and storage ($n + 2\sqrt{n}$ slots) and obviously sorts by (logical address, recency), with dummies and $\perp$ sorted last. The client then linearly scans the sorted list, marking each stale copy of an address as a dummy, keeping only the newest. The client samples a fresh key k' , retags real items as $F_{k'}(i)$ and the first $\sqrt{n}$ dummies as $F_{k'}(n + j)$ (discarding the rest), obviously sorts by tag, and clears the stash. This costs $\tilde{O}(n)$ overhead by Lemma 2.

Cost. Each access reads $\sqrt{n}$ stash slots and $O(\log n)$ storage slots, and writes one slot. Each epoch additionally costs $\tilde{O}(n)$ for the rebuild, amortized over $\sqrt{n}$ accesses. Total amortized overhead per access is therefore $\tilde{O}(\sqrt{n})$, versus $\Theta(n)$ for trivial ORAM.

Exercise 4. *Prove the above protocol realizes ORAM in the UMA-hybrid model.*

Hint: the simulator is given the number of accesses. Steps 1 and 3 and the rebuild touch slot sequences independent of the requests, and a binary search is determined by its target slot, so it suffices to simulate the sequence of target slots. Argue that within an epoch no tag is targeted twice, then replace F_k by a random function.

Next Time

While this construction achieves sublinear overhead, it is still quite expensive. Still, it introduces one of the key ideas underlying ORAM: continually rearrange memory to obfuscate the access pattern. Next time we will see a better construction that achieves only *polylogarithmic* overhead, and we will see more connections between ORAM and secure computation protocols.

References

- [AS96] Laurent Alonso and René Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.
- [Bat68] Ken Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. Association for Computing Machinery, 1968.
- [GHL⁺14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCs*, pages 405–422. Springer, Berlin, Heidelberg, May 2014.
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.