Oblivious RAM, Continued

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2025

Last time, we introduced the notion of Oblivious RAM (ORAM). Recall that we defined ORAM in terms of an interaction between a *client* and a *server*. The client is some low-space machine that wishes to outsource its storage to the powerful-yet-untrusted server. Of course, the client can encrypt its data before sending it to the server, but this is not enough to achieve provable security. Indeed, the server stills observes the client's access pattern into its data, which can leak considerable information about the client's data.

The client can address this access pattern problem by using an *ORAM compiler*. Recall that an ORAM compiler is also a low-space machine (and hence it can run on the client) which translates the client program's *logical* memory addresses into *physical* accesses to the server. The security property of this compiler is that the physical access pattern can be simulated given only the *length* of that pattern. Hence, in the client-server setting, the server learns nothing except the runtime of the client program.

Last time, we constructed a basic ORAM scheme that achieves $\tilde{O}(\sqrt{n})$ *blow-up*. Here, $\tilde{O}(\cdot)$ hides $\log n$ factors, and *blow-up* refers to the ratio between the length of the physical sequence and the length of the logical sequence. Namely, how much more expensive is the client/server interaction after compilation.

Today, we will follow on that discussion in two ways.

- First, we will discuss Path ORAM [SvDS+13], which remains one of the more efficient and influential known ORAM compilers. In particular, Path ORAM achieves $O(\log^2 n)$ blow-up, very considerably better than the scheme we saw last time.
- Second, we will discuss how ORAM can be integrated into an MPC protocol.

Path ORAM

Path ORAM is a surprisingly simple and efficient data structure that achieves oblivious compilation with overwhelming probability. Recall from last time that a main idea in ORAM is that the compiler can continually rearrange memory on the server such that when the a physical address is later queried, the server cannot tell which logical item was accessed. Last time we saw how to do this by periodically shuffling all of memory, but the Path ORAM schematic is considerably simpler.

Path ORAM is a so-called *tree-based ORAM* [SCSL11]. The main idea of this ORAM is that the compiler will organize the client's data into a perfect binary tree with $O(n)$ leaves. Note that the tree therefore has $O(\log n)$ levels. Each node in the tree is a *bucket* that can store up to a small bounded number Z of data items. (In particular, the technique can be proven to work when $Z = 4$.) As an exception, the *root* of the tree will be a larger bucket of size $S = O(\lambda)$; this root is called the stash, and since it is small, we can save it inside the ORAM compiler itself. That is, the stash is not kept on the server.

At all times, each logical memory address will be assigned to some leaf of the tree, which is chosen uniformly at random. The **crucial invariant** of the technique is that each logical memory item will be guaranteed to lie in some node of the tree on the path towards its assigned leaf. Therefore, if the client wishes to download some particular logical address i, it is sufficient for the compiler to (1) determine i's assigned leaf α, (2) download the path through the tree towards α, and (3) search the path for logical address i. By the invariant, this search will indeed yield the desired memory item.

Like we discussed last time, a key challenge in ORAM is simply in remembering the mapping between logical addresses and physical addresses, and the issue is the same here. In particular, the compiler must

remember the mapping between addresses i and leaves α. Unfortunately, this mapping is of size n. Again like we discussed last time, this can be solved by means of a recursion: We store the mapping between logical addresses and physical addresses in a smaller ORAM. This smaller ORAM is typically called the **position map**.

Now we have seen how to download a memory address i, but of course, the client program might want to read memory address i again in the future. Notice that it is *not secure* to keep memory address i assigned to leaf α. Indeed, if we leave i assigned to α, then the server can easily distinguish memory accesses to repeated locations from memory accesses to distinct locations.

Therefore, once the compiler reads address i from path α, it samples a fresh leaf address β uniformly at random, and reassigns i to β. Now, to restore the invariant, we must ensure that logical item i lies on the path to β, but we must place it on that path carefully to ensure obliviousness, since we do not want to server to link memory writes with subsequent reads.

The trick here is to place logical item i in the stash, at the root of the tree. Indeed, *all* paths pass through the root, so there is no way for the server to distinguish which path was written. Of course, the problem with this approach is that currently on each access, we add a new item to the stash, so the stash will continue to grow, and after a relatively small number of accesses, the stash will exceed its size-S capacity.

The solution to this problem is called **eviction**. Roughly, eviction is a strategy by which the compiler continually tries to push (or evict) data items down the tree towards the leaves, where there is more space. The idea is that if we can push items down the tree fast enough, then we can move items out of the stash and into the tree, leaving room for subsequent writes to the stash.

The Path ORAM eviction strategy is based on the following insight: recall that when the compiler performs a read, it downloads an entire path of the tree. This gives an opportunity for the compiler to move items down the tree! Indeed, the compiler can rearrange the downloaded path, then save it back to the server in the rearranged order, where items have been pushed further down.

Formally, the Path ORAM eviction strategy is as follows. First, let us assume that each data item stored on the server is tagged with its assigned leaf address. We say that a data item may **legally reside** in some tree node if its assigned path passes through that node. Now, suppose the compiler downloads path α as part of a read. The compiler then proceeds as follows:

- Concatenate all data items in the stash with all data items on the downloaded path. Let the result be an array A.
- Sort A by the following criteria: An item a is greater than b if a legally resides farther down path α than does b. In other words, upon sorting, the first item in A legally resides at least as far down α as any other item in A.
- Next, the compiler starts from the leaf node of path α and moves towards the root (i.e. the stash). At each step, it performs the following: If the next item in A legally resides in the current node, and if the current node still has available space, then place that item in the current node, and move to the next item in A. Otherwise, move up the path to the next node.

Once this is done, the compiler saves the path back to the server. (It should be noted that other works have attempted different eviction strategies, with different outcomes, e.g. [WCS15, RFK$^+$15].)

Stash Analysis. Amazingly the above eviction pattern can be proven to "work". By this, we mean that the stash will never exceed its capacity, except with negligible probability. The proof of this fact is quite difficult, and beyond the scope of this course. But, it is a nice exercise in some interesting concepts from probability theory. See [RFK$^+$15] for a nice version of the proof.

As one key insight, the proof considers two versions of the ORAM: (1) the ORAM we have seen where nodes have bounded capacity of size Z and (2) a fictional version of the ORAM where nodes have *infinite* capacity. We consider running these two ORAMs with the same randomness and the same access pattern. Using some careful analysis, one can establish a tight correspondence between these two versions of the following kind:

> The bounded ORAM's stash (of size S) overflows *if and only if* there is some *rooted subtree* T of the infinite capacity ORAM that holds more than $S + |T|Z$ items.

This makes it possible to study the infinite capacity ORAM, rather than the bounded one, which is considerably friendlier to probability analysis. Indeed, by using the fact that leaf addresses are assigned uniformly at random, one can conclude that *no* subtree exceeds size $S + |T|Z$, except with probability negligible in S.

Complexity. As stated at the beginning, ORAM achieves blow-up $O(\log^2 n)$. This is quite easy to see:

- On each ORAM access, the compiler downloads a tree path, and paths have length $O(\log n)$. Each bucket on that path has *constant* size, so this is total $O(\log n)$ blow-up.
- Now, to download the correct path, the compiler has to first remember which path to download, which, again, is solved by means of a recursive position map. Utlimately, there are $O(\log n)$ levels of recursion.

Hence, to fetch a single logical item, the compiler fetches $O(\log^2 n)$ physical items.

Can we do better?

At this point, it is natural to wonder if $O(\log^2 n)$ blow-up is the best achievable. In fact, it is not. There does exist an asymptotically better ORAM called OptORAMa [AKL+20] which achieves $O(\log n)$ blow-up. However, this asymptotically-better ORAM has two significant shortcomings:

- First, notice that Path ORAM does not need *any* cryptography to obfuscate the access pattern. In other words, the way the compiler rearranges data is purely combinatorial. In OptORAMa, this is not the case, and compiler must run a PRF to perform its queries. In other words, Path ORAM is a *statistically-secure* ORAM, whereas OptORAMa is only *computationally-secure*. This difference matters greatly when we plug ORAM into MPC.
- Second, OptORAMa incurs impractical constant factors in its concrete cost. In particular, Path ORAM's asymptotics hide a constant factor of roughly two (depending on how the position map is set up), while OptORAMa's asymptotics hide a constant factor of ≈ 9400. [AKM23] shows how to greatly improve OptORAMa's constant factors, but at the cost of blowing up the compiler storage.

Can we hope for an ORAM with less than $o(\log n)$ blow-up? The answer here is no [AKL+20]. There is a provable lower bound that shows that any ORAM scheme necessarily incurs $\Omega(\log n)$ blow-up. Notably, for statistical ORAM, the best schemes all incur $O(\log^2 n)$ blow-up, and it is not clear if this blow-up is necessary or not.

Plugging ORAM into MPC

Recall that we began our discussion of ORAM in search of achieving MPC protocols for arbitrary RAM programs. In fact, it is relatively easy to use ORAM to achieve this goal. In particular, we can consider upgrading the interactive GMW protocol with RAM. The idea is as follows:

- The MPC parties will hold *secret shares* of the server's memory.
- The MPC protocol will implement the client program and the ORAM compiler as a Boolean circuit.

Roughly speaking, we can consider organizing our MPC program as the execution of a basic CPU step. At each step, the CPU manipulates some small number of local registers and ultimately issues a memory request. All of this can be easily achieved via a Boolean circuit. Now, the memory request cannot yet be revealed to the parties, since its content would reveal to the parties information about internal values of the computation. However, if we use an ORAM compiler, the resulting memory queries *can* be simulated. So the parties can simply reconstruct those memory queries, then locally read corresponding secret shares from their memory.

The only question is as follows: How big is the circuit that implements the ORAM compiler? In Path ORAM, it's relatively easy to see that we need a circuit of size roughly $O(\lambda \log^3 n)$, because the ORAM eviction requires us to *sort* an array of size $O(\lambda)$ consisting of $O(\log n)$-bit data items. Note that the Circuit ORAM construction [WCS15] designs a slighly different eviction algorithm with this circuit-based use-case

in mind. Its eviction is implemented by a circuit of size $O(\lambda \cdot \log n)$. This eviction is called $O(\log n)$ times per access, once per level of position map.

Thus, if we have a RAM program that runs in time T, there is a p-party MPC protocol that securely implements that program and that uses $O(p^2 \cdot T \cdot \lambda \cdot \log^2 n)$ bits of communication.

Next Time

A statistical ORAM can be straightforwardly plugged into an interactive MPC protocol, but it is less straightforward how to plug it into non-interactive garbled-circuit based protocols. Next time, we will explore the challenge of so-called Garbled RAM, which shows how to integrate RAM into garbled circuits.

References

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Cham, May 2020.

[AKM23] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. FutORAMa: A concretely efficient hierarchical oblivious RAM. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 3313–3327. ACM Press, November 2023.

[RFK+15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, August 2015.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Berlin, Heidelberg, December 2011.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.