1 Lecture 18: Oblivious Ram (ORAM)

Consider how google drive and dropboxes work. Your information is stored in the cloud, and whenever you need it, you access it. However, the cloud can see your access patterns for the files (i.e., when it says "you accessed this file 3 times today", or "you usually open this file around this time"). In general, this isn't that big of a concern, as long as they share this only with you. But what if someone knew you were hospitalized a couple times, on specific dates? Then, if they had access to the hospital's cloud access patterns, they could figure out which hospital record is yours and try to get ahold of it. And even if they couldn't get ahold of it, they could still figure out any other time that you are sick/hospitalized, since the hospital would access your record again.

ORAM hides access patterns to (encrypted) files.

We might intuitively define "oblivious" based on indistinguishability: Any two sequences of read/write commands should be indistinguishable.

More formally, we could play a game between a challenger C and adversary A:

1. A sends a database to C
2. A sends two access patterns S_0 and S_1.
3. C encrypts the database
4. C picks a random bit b and simulates the accesses S_b to the encrypted database
5. C sends the sequence of simulated accesses to the encrypted database
6. A tries to pick the bit, and wins if it can do so with non-negligible advantage over 1/2.

We start with a special case, where both access patterns are just permutations of the entire database. The user might encrypt the database by shuffling items around, in which case the encryption of the database would be a permutation itself. So any access pattern the challenger uses would be another permutation of the database, and the adversary would have no way of telling which bit was chosen.

But, as noted, this is a special case, and it is easier than the normal case for the challenger to guarantee security here. What if S_0 is a bunch of accesses to the same spot, and S_1 is a bunch of accesses to different spots? This is obviously an issue for our current strategy, because even if we shuffle around the database entries, the adversary can easily tell the difference between multiple accesses to the same spot and several accesses to different spots.

Another special case is where the user has enough local memory to store the whole database. Then, it will never need to do repeated accesses, so everything becomes a permutation (we can do fake accesses to spots we don't need). But this is not an interesting case.

So, we focus on when the user has small memory (ideally, $O(1) \cdot poly(k)$). What if the user has memory $O(\sqrt{n})$, where n is the size of the database? As alluded to before, we can cache a lot of values, and instead of making a double-access to a value that is accessed twice in the access pattern, we can fake an access to some other location that we don't need. We keep doing this until we are out of space in our cache, then we rearrange all of the entries of the database and store our previous cache info in the cloud.