Notes for Lecture 13

Last time we constructed iO for branching programs. This allowed us to apply Barrington's Theorem and get iO for NC^1 (languages with log-depth circuits). Today we will see something called boosting to get iO for $\mathsf{P/poly}$ (languages with poly-size circuits).

1 iO for $\mathsf{P/poly}$

Claim 1. *If we have iO for NC^1 as well as Fully Homomorphic Encryption where we can decrypt in log-depth (we denote this as FHE^*), we can get iO for $\mathsf{P/poly}$.*

The construction is as follows. Suppose we have iO for NC^1, and we have an FHE^* scheme $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}$.

We construct $\mathsf{iO}'(\cdot)$ that can obfuscate any poly-size circuit C. First, generate

$$(dk_0, ek_0, \oplus_0, \otimes_0) \leftarrow \mathsf{Gen}(), (dk_1, ek_1, \oplus_1, \otimes_1) \leftarrow \mathsf{Gen}().$$

Set $d_0 \leftarrow \mathsf{Enc}(ek_0, C)$ and $d_1 \leftarrow \mathsf{Enc}(ek_1, C)$. Assume we have some circuit $F_x(\cdot)$ that computes $F_x(C) = C(x)$. Using this, we can compute $\mathsf{Enc}(ek_b, C(x))$ by applying the homomorphic operations to d_b.

Let $\mathsf{Eval}_b(x, d_b)$ be the program that applies the homomorphic operations to evaluate C on x. π will be an NP proof that $e_0 = \mathsf{Eval}_0(x, d_0)$ and $e_1 = \mathsf{Eval}_1(x, d_1)$. The proof π can be generated by running $\mathsf{Eval}_b(x, d_b)$ for $b = 0, 1$ and writing down the values on all the wires. Verifying these gates can be done in log depth.

So now we define $P_{dk_0}(e_0, e_1, x, \pi)$ as the circuit the first checks if π is a valid proof that $e_0 = \mathsf{Eval}_0(x, d_0) \wedge e_1 = \mathsf{Eval}_1(x, d_1)$. If so, it outputs $\mathsf{Dec}(dk_0, e_0)$. Otherwise it aborts.

Finally, our iO' scheme will output $(\mathsf{iO}(P_{dk_0}), \mathsf{Eval}_0, \mathsf{Eval}_1, d_0, d_1)$.

To recap: To use this iO' scheme, we would input a poly-sized circuit C and then receive $(\mathsf{iO}(P_{dk_0}), \mathsf{Eval}_0, \mathsf{Eval}_1, d_0, d_1)$. To evaluate C on x, we would compute $e_0 = \mathsf{Eval}_0(x, d_0), e_1 = \mathsf{Eval}_1(x, d_1)$. Then we generate the proof π by examining the wires of these circuits, and then we feed (e_0, e_1, x, π) into $\mathsf{iO}(P_{dk_0})$, which will output $C(x)$. Note that all the operations of P_{dk_0} can be done with a log-depth circuit.

Theorem 2. *The above iO scheme is secure.*

Proof. Suppose we have $C_0 \equiv C_1$.

Hybrid 0. We set $\hat{P} = iO(dk_0), d_0 = \mathsf{Enc}(ek_0, C_0), d_1 = \mathsf{Enc}(ek_1, C_0)$.

Hybrid 1. This is the same as Hybrid 0, except we change d_1 so that $d_1 = \mathsf{Enc}(ek_1, C_1)$. This is indistinguishable from Hybrid 0 by FHE security, since ek_1 is hidden.

Hybrid 2. This is the same as Hybrid 1, except we change the program to be $\hat{P} = iO(P_{dk_1})$. The only change is in the last line where we change it to output $\mathsf{Dec}(dk_1, e_1)$. The claim is that P_{dk_0} is equivalent to P_{dk_1}. To see why, suppose not. The two programs will only output something at the same inputs (since the checks are the same), so they must differ at some point where they both produce an output. So we have $P_{dk_1}(e_0, e_1, x, \pi) \neq P_{dk_0}(e_0, e_1, x, \pi)$ for some e_0, e_1, x, π. But since they both output something, the check passed. This means we must have $e_0 = \mathsf{Eval}_0(x, d_0)$ and $e_1 = \mathsf{Eval}_1(x, d_1)$. So $e_0 = \mathsf{Enc}(ek_0, C_0(x))$ and $e_1 = \mathsf{Enc}(ek_1, C_1(x))$. P_{dk_0} outputs $C_0(x)$ and P_{dk_1} outputs $C_1(x)$, which are equivalent since $C_0 \equiv C_1$. This directly contradicts our assumption. Since the programs are equivalent, iO security (of our underlying NC^1 iO scheme) implies Hybrid 1 is indistinguishable from Hybrid 0.

Hybrid 3. This is the same as Hybrid 2, except we change d_0 to be $d_0 = \mathsf{Enc}(ek_0, C_1)$. This is indistinguishable from Hybrid 2 by FHE security, since now ek_0 is hidden.

Hybrid 4. We change the obfuscated program back to $\hat{P} = iO(P_{dk_0})$. This follows from the same proof as given for Hybrid 2.

Hybrid 0 is equivalent to the C_0 case and Hybrid 1 is equivalent to the C_1 case, so this completes the proof. □

2 iO for $\mathsf{P/poly}$ from Weaker Primitives

We can give a more flexible approach that uses weaker primitives than FHE^*.

Theorem 3. *If we have subexponentially secure* iO *for* NC^1 *and PRFs in* NC^1*, we can get* iO *for P/poly.*

Let's recall the way we obfuscated matrix branching programs (MBP) in the previous lecture.

An MBP is composed of two sequences of l square matrices $A_{i,b}$ for $i \in [l]$ and $b \in \{0, 1\}$. We can think of these as being written out as two rows, so that the matrices $A_{i,0}, A_{i,1}$ are in the ith column. Additionally, we have a bookend row vector s at the beginning and a bookend column vector t at the end. Finally, we have a function $inp(i) : [l] \to [n]$. On input $x \in \{0, 1\}$, $M(x)$ evaluates to $s \cdot (\prod_{i=1}^{l} A_{i,x_{inp(i)}})t$. We say that $BP(x) = 1$ iff $M(x) = 0$.

We applied Killian rerandomization which takes each $A_{i,b}$ and replaces it with $\alpha_{i-1,b}R_{i-1}^{-1}A_{i,b}R^i$. We also replaced s with $\alpha_0 s R_0$ and t with $\alpha_{l+1}R_l^{-1}t$. We then encoded these matrices in an asymmetric multilinear map, thinking of each matrix as a separate plaintext element. The idea was that if we multiplied one matrix from every column (as we would with an honest execution), this would correspond to an element in the top level of the multilinear map, which we could zero test.

The problem was that the naive implementation of this, which was to associate the singleton set $\{i\}$ with column i, is vulnerable to a mixed-input attack (see Lecture 12 notes for an example). We fixed this with straddling sets. These are two sequences of sets $S_0^{(b)}, \ldots, S_j^{(b)}$ for $b = 0, 1$ such that $\cup_j S_j^{(0)} = \cup_j S_j^{(1)}$. We also need $S_i^{(b)} \cap S_j^{(b)} = \emptyset$ for $i \neq j$. We also need that $\forall T_0, T_1 \neq [j]$, we have $\cup_{j \in T_0} S_j^{(0)} \neq \cup_{j \in T_1} S_j^{(1)}$. We can do this with $S_0^{(0)}, \ldots, S_j^{(0)}$ set to $\{0\}, \{1, 2\}, \ldots, \{j-1, j\}$ and $S_0^{(1)}, \ldots, S_j^{(1)}$ set to $\{0, 1\}, \{2, 3\}, \ldots, \{j-2, j-1\}, \{j\}$.

For each value of $i \in [l]$ we come up with a different straddling set system disjoint from all the other set systems. We encode the elements for an input bit in the levels corresponding to these sets.

So our matrices are encoded as $[\alpha_{i,b}R_{i-1}^{-1}A_{i,b}R_i]_{S_k^{(b)}(inp(i))}$, where $inp(i)$ specifies which straddling set system we are using.

This obfuscator gives iO security in the generic multilinear map model, but last time we claimed that this model isn't great because we can get VBB security.

Let's suppose we're in the setting where each input bit is read only once. In that case we don't need to worry about straddling sets. What we can do is multiply the sum of the two matrices of the first column with the sum of the two matrices of the second column and so on for every column. This comes out to look like $\sum_x M(x)$ (with some unimportant α factors in front, which are not important for this illustration). This sum is 0 if and only if $BP(x) = 1$ everywhere. By making this query, I can learn whether or not this function is identically 1, which is something that I should not be able to do with VBB obfuscation.

To address this issue, we'll use dual input branching programs. This is where each column has four matrices instead of two. For our notation, the first subscript will denote column, and the last two subscript bits will specify one of the four matrices. So for example column 3 consists of matrices $A_{3,0,0}, A_{3,0,1}, A_{3,1,0}, A_{3,1,1}$. The $inp(\cdot)$ function now outputs $\{j, k\}$, and we'll write $inp_0(i) = j$ and $inp_1(i) = k$. So now the program evaluates to

$$M(x) = s \cdot \prod_{i=1}^{l} A_{1, x_{inp_0(i)_1}, x_{inp_1(i)}} \cdot t.$$

We'll assume each pair $\{j, k\}$ is read at least once. If our original branching program were sufficiently long, then we can make this happen. If the branching program is not

long enough, then we can pad it with extra matrices. Now each matrix is encoded as

$$[\alpha_{i,b_0,b_1} R_{i-1}^{-1} A_{i,b_0,b_1} R_i]_{S_{k_0}^{(b_0)}(inp(i)_0) \cup S_{k_1}^{(b_1)}(inp(i)_1)}$$

Correctness is not hard to see.

Theorem 4. *In the generic multilinear map model, for any isZero query on $[P]_{[k]}$. We can write $P = \sum_{x \in S} P_x$ where $|S|$ is polynomial size and P_x depends on encodings consistent with x.*

It turns out by querying $C(x)$ $\forall x \in S$ we can decide if $P = 0$. This allows us to simulate the view of the adversary.