Notes for Lecture 12

1 Overview

Today, we'll see how to use multilinear maps to build obfuscation. We'll build obfuscation for a restricted model of computation called branching programs, so we won't get circuit obfuscation right away.

Using Barrington's Theorem, we can build iO for NC^1. This class contains all functions computable by log depth circuits. It is essentially corresponds to boolean functions. With boosting/bootstrapping and FHE, we can get iO for polynomial size circuits.

2 Matrix Branching Programs

In a matrix branching program (MBP), we have two sequences of l square matrices. Imagine writing out these matrices in two rows, such that the first row contains the matrices $A_{i,0}$ for $i \in [l]$ and the second row contains the matrices $A_{i,1}$ for $i \in [l]$.

Matrix branching programs are evaluated on inputs $x \in \{0,1\}^n$. We will have a function $inp : [l] \to [n]$ that maps column indices to n. So we can imagine associating each column i with $inp(i)$. We say that the width of an MBP is the dimension of the square matrices $A_{i,b}$.

On input x, the $M(x)$ evaluates to

$$M(x) := \prod_{i=1}^{l} A_{i,x_{inp(i)}}.$$

An MBP gives us the boolean function $F(x)$, which is 0 if $M(x) = I$ and 1 otherwise.

So how do we get such a program for a log depth circuit?

Theorem 1. *(Barrington's Theorem) For a circuit C of depth d, we can get a matrix branching program BP with width 5 and $l = 4^d$.*

We can only handle NC^1 because the size of the branching program blows up with large depth.

As expected, Barrington's Theorem gives us an MBP where $M(x)$ will be I if $C(x) = 0$ and some other fixed matrix A if $C(x) = 1$.

Proof of Barrington's Theorem.

We will use matrices A, B such that $A, B, ABA^{-1}B^{-1}$ are all pairwise similar. A and B are similar if $A = PBP^{-1}$ for some invertible matrix P. This holds trivially for A and B set to the identity matrix, so we require that they not be the identity.

Suppose I want to get a matrix branching program that computes some circuit. Assume by induction that it can compute the wire(s) leading into the top gate. We consider the possible cases for the top gate.

If it is a NOT gate, with $M_0(x)$ being the input MBP, then we just set $M(x) = PA^{-1}M_0(x)P^{-1}$. To implement this, we take the two leftmost matrices of the $M_0(x)$ branching program and multiply them on the left by PA^{-1}, and we take the rightmost matrices of $M_0(x)$ and multiply them on the right by P^{-1}. So if $M_0(x) = A$, then $M(x)$ outputs the identity. If $M_0(x) = I$, then the output is $PA^{-1}P^{-1}$.

Suppose it is an AND gate with $BP_0(x)$ and $BP_1(x)$ fed in with respective MBPs $M_0(x), M_1(x)$. We use the fact that we have matrices A and B that are similar. We have $M_0(x)$ output A or I and $M_1(x)$ output B or I. Set $M(x) = M_0(x)M_1(x)M_0^{-1}(x)M_1^{-1}(x)$. We can implement this by concatenating four branching programs. If either $M_0(x)$ or $M_1(x)$ is I, then clearly this collapses to I. But if both are not I, then this becomes $ABA^{-1}B^{-1}$.

3 Obfuscating Branching Programs

We'll now consider the "bookend" version of a matrix branching program. To the left of all the matrices, we add a $1 \times w$ row matrix s and to the right of all the matrices, we add a $w \times 1$ column matrix t (w is the width the of the MBP). So the MBP product is just a scalar

$$M(x) := s(\prod_{i=1}^{l} A_{i,x_{inp(i)}})t.$$

We define Killian rerandomization to work as follows. Pick random matrices of dimension $w \times w$ random matrices R_i for $i \in [l]$ and random scalars α_{ib} for $i \in l$ and $b \in \{0, 1\}$. Also pick two bookend random scalar α_0 and α_{l+1}. For each matrix A_{ib}, left multiply by $\alpha_{ib}R_{i-1}$ and right multiply by R_i. For bookend s, right multiply by $\alpha_0 R_0$. For bookend t, left multiply by $\alpha_{l+1}R_l^{-1}$.

When we compute $M(x)$, these random matrices will cancel each other out and the input/output behavior should be unaffected. The randomization should ideally obfuscate everything. The problem is that we don't actually get any security from this.

It turns out that the right way to do this is by encoding everything with a multilinear map. If we actually multiply a matrix from every column (corresponding to an honest execution), this should correspond to reaching the top level of the multilinear map. At this point, we can zero test.

Ideally, we want that some simple assumption, such as the Multilinear Decisional Diffie Hellman assumption, implies that this construction gives us iO. It turns out this won't work. For starters, the MDDH assumption does not appear strong enough for us. Moreover, as we will see in the coming lectures, there are attacks on multilinear maps that render many of these simple assumptions false.

What we'll do instead is argue security in the Generic Multilinear Map Model. In this model, we assume that the adversary only interacts with the multilinear map in legal ways.

The model works as follows. We have some adversary A. The model M is given some plaintext elements $a_1, \ldots, a_m$. We can imagine these correspond to the Killian rerandomization matrices before we encoded them. The model is also given what level to encode these plaintexts elements at. For now, all it does is it writes these down.

It comes up with random string labels $\{L_i\}_{i=1,\ldots,m}$. When the adversary wants to add $L + L'$, the adversary looks for a, S, a', S' It checks that $S = S'$. Then it creates $a'' = a + a'$, and creates a new label L'' for a''. We can define multiplication similarly. When the adversary querys on $isZero(L)$, it checks if the corresponding S is actually equal to $[k]$ and then responds with whether or not a is 0.

Theorem 2. *We can construct VBB obfuscation in the Generic Multilinear Map Model.*

How is this possible? After all, in the first lecture, we showed that VBB obfuscation does not exist! What this says is that there must be attacks on obfuscation that lie outside of the generic multilinear map model. If you think about our attack from the first lecture, this actually makes sense. In that attack, the program is evaluated homomorphically on some input. Our homomorphic encryption scheme can homomorphically evaluate any function that can be represented as a circuit of, say, $\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}$ gates. However, in the generic multilinear map model, evaluating the obfuscated program involves making calls to the model itself. This means it is not possible to represent the program evaluation as a circuit in this model, and we can therefore not homomorphically evaluate the obfuscated program.

It actually turns out that many generic models suffer from weaknesses such as this. However, the attacks on this tend to be pathological, so we hope that in most normal cases we can still get security.

How do we encode levels? The obvious way is to have the singleton set $\{i\}$ correspond to column i. The problem is that the adversary can do a mixed input attack. Suppose

for example that $inp(1) = inp(4) = b$. Then our branching program must include $A_{1,b}$ and $A_{4,b}$. In other words, we must pick the top matrix in both columns 1 and 4 or the bottom matrix. But nothing is stopping the adversary from picking the top matrix in column 1 and the bottom matrix in column 4. These attacks are dangerous because two equivalent programs are only equivalent when you are evaluating them on normal inputs.

We can make this program robust against such attacks by using straddling sets. Continuing the above example, if we assign $\{1, 3\}$ to the top matrix of column 1 and $\{1, 4\}$ to the bottom matrix of column 1, and $\{2, 4\}$ to the top matrix of column 4 and $\{2, 3\}$ to the bottom matrix of column 4, we won't be able to mix inputs. Any attempt to do so will cause an element in the sets to be in common, which is not allowed when multiplying in asymmetric MMaps.