COM S 6830 – Cryptography November 19, 2009

Lecture 23: HBC and Oblivious Transfer

Instructor: Rafael Pass *Scribe: Eleanor Birrell*

1 Honest but curious adversaries

Previously, we have considered secure computation with respect to arbitrary adversaries. Now, we temporarily restrict our attention to Honest-but-Curious adversaries (HBC), that is adversaries that run the protocol correctly until the last step, during which they can perform any (poly-time) computation and then output the resulting string. Why is this a good model? HBC can be enforced with trusted hardware or zero-knowledge proofs.

1.1 1/4-Oblivious Transfer

Definition 1 *A (1/4)-Oblivious Transfer is a 2-party computation of the function*

$$F((a_1, \ldots, a_4), b) = (\perp, a_b).$$

Theorem 2 *If there exist trapdoor permutations over $\{0,1\}^n$, then there exists an HBC (1/4)-OT.*

Proof. Consider the following protocol:

Step	$A(a_1, \ldots, a_n, 1^n)$		$B(b, 1^n)$
1	$(i, tk) \leftarrow Gen_f(1^n)$		
	i	$\rightarrow$	
2			$x_j \leftarrow \{0,1\}^n : y_{j \neq b} = x_j, y_b = f_i(x_b)$
		$\leftarrow$	$(y_1, \ldots, y_4, h)$ (h is a hard-core predicate for f_i)
3	$z_j = h(f_i^{-1}(y_j)) \oplus a_j$		
	$z_1, \ldots, z_4$	$\rightarrow$	

We claim that the given protocol is a (1/4)-Oblivious Transfer. For b, $z_b = h(f_i^{-1}(f(x))) \oplus a_b$ therefore B can efficiently compute a_b. For $j \neq b$, in order to compute a_j, B must compute $h(f_i^{-1}(y_j))$. However, since f is a permutation, the distributions of y_j and $f_i^{-1}(y_j)$ are both uniformly random, so this is equivalent to evaluating the hardcore predicate of a random input given its image under a one-way permutation, which by definition is infeasible. Therefore B gets output a_b. A only receives $y_1, \ldots, y_4$ and h. Since f is a one-way permutation, $\{0,1\}^n$ and $f(\{0,1\}^n$ are identically distributed, therefore A's view is indistinguishable from four randomly chosen values.

Problem: How does A know that B isn't giving four values of the form $f(x_i)$ for known values x_i instead of one value of that form and three randomly chosen values? We cannot. However, since we are only considering HBC adversaries, this does not affect the security of the protocol. As we will see in the next class, this plus ZK is sufficient to gain protocol secure against arbitrary adversaries.

Using Oblivious Transfer, we can now construct a HBC protocol that computes any efficiently computable function.

Theorem 3 *For any poly-time computable function $f : (\{0,1\}^n)^{m(n)} \leftarrow \{0,1\}$, there exists a protocol that securely computes f with respect to HBC adversaries.*

Proof. In fact, we will show a compiler that given f outputs a communication protocol $P_1, \ldots, P_n$ that describes a secure computation with respect to HBC adversaries.

f is efficiently computable, so there exists an Turing machine f in polynomial time. Therefore by Cook reduction, can be computed by a circuits composed of NOT and AND gates. Each gate can be viewed as a finite operation over GF_2.

$$NOT(x) = x + 1$$
$$AND(x, y) = xy$$

High-level: The first step is that the parties are going to "share" their inputs using (m, m)-secret sharing. Then gate-by-gate compute on shares, that is given shared inputs MPC perfectly random shares of output. Final step is reveal all shares.

We already know how to share secrets, so we just need to show how to deal with the two types of gates: not and and.

Not-gates: Assume (by induction) that all parties have random shares $b_1, \ldots, b_n$ respectively of a value $b = \sum b_i$. we now want to compute a random sharing of $\neg b$. Define new shares b'_i such that $b'_1 = b_1 + 1$ and $b'_i = b_i$ else. By induction, we still have a random sharing as an output.

XOR-gates (just for fun!): Assume (by induction) that all parties have random shares $c_1, \ldots, c_n$ and $d_1, \ldots d_n$ and want to compute random sharing of $b = c + d$. Define $b_i = c_i + d_i$.

AND-gates: Assume (by induction) that all parties have random shares $c_1, \ldots, c_n$ and $d_1, \ldots d_n$ and want to compute random sharing of $b = cd$. Observe that $cd = \sum c_i \cdot \sum d_i = \sum c_i d_i + \sum_{i<j} (c_i d_j + c_j d_i)$.

For each $j > i$, P_j sends to P_i a (1/4)-OT of the values $(b_{j,i}, c_j, d_j, c_j + d_j)$ where $b_{j,i}$ is chosen at random. P_i chooses as follows

$$\begin{aligned} a_1 = b_{j,i} \quad & if c_i = 0, d_i = 0 \\ a_2 = c_j \oplus b_{j,i} \quad & if c_i = 0, d_i = 1 \\ a_3 = d_j \oplus b_{j,i} \quad & if c_i = 1, d_i = 0 \\ a_4 = c_j + d_j \oplus b_{j,i} \quad & if c_i = 1, d_i = 1 \end{aligned}$$

Observe that in all cases, $a_b = c_i d_j + c_j d_i + b_{j,i}$; define this to be $b_{i,j}$, We then define $b_{i,i} = c_i d_i$ and define the shares $b_i = \sum_j b_{i,j}$. Correctness follows immediately. Moreover, each value $b_{j,i}$ (for $j > i$) is chosen uniformly at random, therefore a_b is a uniformly random value for all b, so the new sharing is private and random.

The round-complexity of this algorithm depends on the depth of the circuit. There is an alternative construction (by Yao) based on garbled circuits that provides a constant-time computation for any two-party function.

2 From HPC to Malicious Security

The main idea is to use ZK to prove to other parties that you are doing whatever you were supposed to be doing. Consider the following approach (which almost works).

1. Commit to input, randomness
2. For every message, send message and ZK proof that message correctly computed.

The problem is that this allows each party to choose its own randomness, thereby allowing an adversary to manipulate the output. To solve this problem, we introduce a general form of coin tossing. Each person tosses coin, puts in envelope, exchange, open up and xor bits. Another version, known as coin tossing into a well, simply eliminates the decommitment step, which enables zero-knowledge enforcement of HBC adversaries in the following manner.

1. Commit to input, post $C(b_i)$, b'_i
2. For every message, send message and ZK proof of knowledge of input, $\{b_i\}$ consistent with message