CS 282A/MATH 209A: Foundations of Cryptography — Prof. Rafail Ostrovsky

Lecture 10

Lecture date: 14 and 16 of March, 2005 — *Scribe: Ruzan Shahinian, Tim Hu*

1 Oblivious Transfer

1.1 Rabin Oblivious Transfer

Rabin oblivious transfer is a kind of formalization of "noisy wire" communication. The objective is to simulate a random loss of information. Formally, a Rabin OT machine models the following behavior. Sender sends a bit b into the OT machine. The machine then flips a coin, and with probability $1/2$ sends b to Receiver, and with probability $1/2$ sends '#' to Reciever to signify that a bit was sent, but the information was lost in the transfer. Sender does not know which output Receiver received.

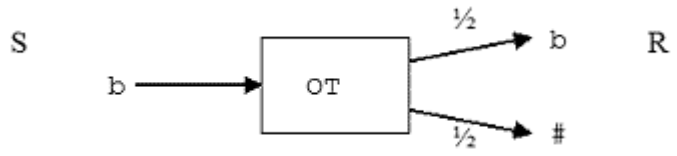

Figure 1: Rabin oblivious transfer

Remark: Note that this may be simulated by a sufficiently noisy wire, provided that the wire transmits faithfully a good proportion of bits and at the same time loses a good proportion of bits, replacing them with noise that is distinguishable from information.

1.2 One-Out-of-Two Oblivious Transfer (1-2-OT)

Even, Goldreich and Lempel formulated a notion of oblivious transfer that has proven useful in various applications. In this situation, Sender sends an ordered pair of bits (b_0, b_1) into the 1-2-OT machine. Receiver then gives the machine a bit i, indicating which input he would like to receive. The machine outputs b_i and discards b_{1-i}. Sender knows that Receiver has one of the bits, but not which one.

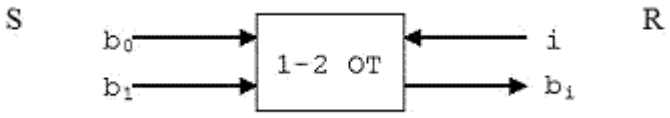

Figure 2: one-out-of-two oblivious transfer

1.3 Implementing Oblivious Transfer Using One Out of Two Oblivious Transfer

The two games described above are information theoretically equivalent, as we will see in the following two sections.

Given a 1-2-OT machine as a black box, the protocol for implementing Rabin oblivious transfer is as follows. Here we only show the reduction for honest players.

1-2-OT $\Rightarrow$ Rabin OT

1. **S** has a bit b which he wants to transmit with a 1/2 probability to **R**.
2. **S** flips random bits r and l.
3. **S** inputs $b_l := b$ and $b_{1-l} := r$ into the 1-2-OT machine. I.e. if $l = 0$, **S** inputs (b, r). If $l = 1$, **S** inputs (r, b).
4. [1-2-OT][1] **R** specifies a bit i to the 1-2-OT machine. Note that the 1-2-OT machine outputs b if and only if $i = l$.
5. [**S** $\rightarrow$ **R**] **S** sends the value of l to **R** in the clear.

After this transfer, Receiver will compare the value of i he picked and the value of l that was sent to him. If $i = l$, then he knows that the bit he received from the 1-2-OT machine was b. If $i \neq l$, then he knows that he was passed the random bit r; in other words, he received no information about b. So Receiver has exactly 1/2 probability of receiving the intended bit.

[1]Square brackets indicate where an exchange takes place.

1.4 Implementing One Out of Two Oblivious Transfer Using Oblivious Transfer

Given a Rabin OT primitive as a black box, the protocol for implementing a one-out-of-two oblivious transfer is as follows. Again, we only show the reduction for honest players.

Rabin OT $\Rightarrow$ 1-2-OT

1. [OT] **S** inputs a large (say, length $3n$) string of random bits $\vec{s}$ into the OT machine, which relays the bits to **R**, replacing approximately half of them with '#'.
2. [**R** $\rightarrow$ **S**] **R** sends to **S** two sets of disjoint indices $I_0, I_1 \subseteq \text{dom}\,(\vec{s})$ chosen at random satisfying:
   (a) I_0, I_1 are of size n.
   (b) One of the sets corresponds to a random subset of places in $\vec{s}$ where **R** received perfect information, i.e. no '#'s. The index of this set (either I_0 or I_1) acts as the i in the description of 1-2-OT transfer.
   (c) The other set is chosen at random.
3. [**S** $\rightarrow$ **R**] **S** chooses the two bits (b_0, b_1) that he would like to send by 1-2-OT, and sends to **R** $(b_0 \bigoplus_{i\in I_0} s_i, b_1 \bigoplus_{i\in I_1} s_i)$.

Note that in step 1, by the Chernoff bound, the probability that Receiver received less then n or more than $2n$ many '#'s is exponentially small. Therefore, except for an exponentially small number of trials, in step 2 it is possible for Receiver to find an index set satisfying (b), and the set chosen in (c) must contain at least one '#'. Thus he knows exactly one of $\bigoplus_{i\in I_0} s_i$ and $\bigoplus_{i\in I_1} s_i$, and he can calculate exactly one of (b_0, b_1).

1.5 Implementation of 1-2 OT

Alternatively we may implement oblivious transfers from cryptographic assumptions. First we will need the notion of an enchanced function.

Definition 1 *A function $f : X \rightarrow Y$ is* enchanced *if there is a polynomial time sampling algorithm that samples Y with the same distribution as f, but for this sampling algorithm, it is hard to invert f.*

Note that for f a one-way permutation an enchancing algorithm is immediate, by picking y at random. Now, the protocol.

1. **S** fixes an enchanced trapdoor permutation f for which he knows the inverse, and a hard-core bit operation HCB for f.
2. [**S** $\rightarrow$ **R**] **S** sends f and HCB to **R**.
3. [**R** $\rightarrow$ **S**] Depending on the selection bit i, **R** takes a random x_i and computes $y_i = f(x_i)$. It also randomly generates $y_{1-i} \in \text{ran } f$ (this is possible since f is an enchanced trapdoor permutation). Then **R** sends back y_0 and y_1.
4. [**S** $\rightarrow$ **R**] **S** then computes $x_0 = f^{-1}(y_0)$ and $x_1 = f^{-1}(y_1)$, and sends to **R** $b_0 \oplus HCB(x_0)$ and $b_1 \oplus HCB(x_1)$.

Now, since **R** knows x_i, it can compute $HC(x_i)$, and then compute b_i. Assuming that **R** was honest in step 3, and really chose y_{1-i} without knowing the inverse, it cannot compute the inverse x_{1-i} and hence its hard-core bit. Thus **R** cannot compute b_{1-i}. Because **S** does not know the selection bit i, he has no idea whether **R** got b_0 or b_1.

Now suppose **R** cheated by not obtaining y_1 randomly, but instead chose an x_1 at random then computing $y_1 := f(x_1)$. Since f is a uniform distribution, **S** would have no way of detecting such behavior. **R** would then in the end know both b_0 and b_1. The protocol works if **R** is "honest but curious," but what if he is "malicious?" The solution to this problem will be explained later in this lecture. For now let us work in an honest-but-curious model.

2 Two Party Secure Computation

Consider the plight of two millionaires who want to find out who is richer, without letting each other know how much money they have. This is the problem of two party secure computation. In a two party secure computation, two parties A and B want to cooperatively run an algorithm where neither party has a complete set of paramaters.

We are given A, B, and a polynomial size circuit $f(\vec{a}, \vec{b})$ consisting of AND and XOR gates which they would like to compute. The problem is that only A has access to the first half of the input (say, $\vec{a}$) and only B has access to the second half of the input ($\vec{b}$). How will they compute $f(\vec{a}, \vec{b})$ without sharing knowledge of $\vec{a}$ and $\vec{b}$? We will build a protocol such that at every intermediate stage of the computation of f, A and B will have a "share" of the output of that stage. The value of each wire is represented as two bits, one bit held by A and the other by B such that their XOR is the value of the wire. Initially, inputs held by A will be split into two such bits for each input bit, where A gives to B a share, and B does the same with its inputs. Now they have to compute the circuit consisting of $\oplus$ and $\cdot$ gates, maintaining the secrecy.

Formally, A will have a record of bits T^A used in the computation and B will have a similar record T^B such that the actual bits used in the computation of $f(\vec{a}, \vec{b})$ are $\{x^A \oplus x^B : x^A \in$

$T^A; x^B \in T^B\}$. Moreover A and B will never be required to reveal information about their shares T. This is done as follows.

Initialization

1. A generates a random string $\vec{a^B}$ and computes $\vec{a^A} := \vec{a} \oplus \vec{a^B}$.
2. $[A \to B]$ A sends $\vec{a^B}$ to B.
3. B generates a random string $\vec{b^A}$ and computes $\vec{b^B} := \vec{b} \oplus \vec{b^A}$.
4. $[B \to A]$ B sends $\vec{b^A}$ to A.

XOR gates: A and B have some $x = x^A \oplus x^B$ and $y = y^A \oplus y^B$, where A knows x^A and y^A, and B knows x^B and y^B, and they wish to compute $x \oplus y$. But since $x \oplus y = (x^A \oplus x^B) \oplus (y^A \oplus y^B) = (x^A \oplus y^A) \oplus (x^B \oplus y^B)$, each party can compute their own share of the sum without cooperation from the other party.

1. A computes $(x \oplus y)^A := x^A \oplus y^A$.
2. B computes $(x \oplus y)^B := x^B \oplus y^B$.

AND gates: A and B have some $x = x^A \oplus x^B$ and $y = y^A \oplus y^B$ and they wish to compute $x \cdot y$.

$$\begin{aligned} x \cdot y &= (x^A \oplus x^B) \cdot (y^A \oplus y^B) \\ &= (x^A \cdot y^A) \oplus (x^B \cdot y^A) \oplus (x^A \cdot y^B) \oplus (x^B \cdot y^B) \end{aligned}$$

Now A can compute $x^A \cdot y^A$ and B can compute $x^B \cdot y^B$, but without revealing their share of x and y, they must compute $x^B \cdot y^A$ and $x^A \cdot y^B$. This is done by oblivious transfer. Let M be a 1-2-OT machine. We first handle the case of $x^A \cdot y^B$.

1. A generates a random bit r^A.
2. A inputs the pair $((x^A \cdot 0) \oplus r^A, (x^A \cdot 1) \oplus r^A)$ to M.
3. B inputs y^B to M.
4. [1-2-OT] M outputs $(x^A \cdot y^B) \oplus r^A$ to B, who stores this as w^B.

Note that $x^A \cdot y^B = r^A \oplus w^B$. Also, B does not get any information from A about x^A, and A does not get any information from M about y^B. The case $x^B \cdot y^A$ is done similarly.

1. B generates a random bit r^B.
2. B inputs the pair $((x^B \cdot 0) \oplus r^B, (x^B \cdot 1) \oplus r^B)$ to M.
3. A inputs y^A to M.
4. [1-2-OT] M outputs $(x^B \cdot y^A) \oplus r^B$ to A, who stores this as w^A.

Finally, A and B can assemble their shares.

1. A computes $(x \cdot y)^A := (x^A \cdot y^A) \oplus r^A \oplus w^A$.
2. B computes $(x \cdot y)^B := (x^B \cdot y^B) \oplus r^B \oplus w^B$.

Lastly, when they compute the output of the "output" wire of the circuit, they can combine their shares and learn the output of the function.

3 Coin Flip Into the Well

We wish to have Alice assign a random bit to Bob over which he has no control; yet, Alice should have no knowledge of the bit. In real life, we might ask Bob to stand beside a deep well, deep enough to be inaccessible to Bob, but shallow enough that the bottom is still visible. Alice will stand from some distance away and toss a coin into the well for Bob, but she is not allowed near the well. Now, it is true that, unless the coin has some magical power of its own, Bob may simply lie about the outcome of the coin toss, as Alice would (and should) never know. Let us suspend this concern until later; first we model this game in practical terms. We will give the coin magical powers later.

Let c be a predetermined commitment scheme.

1. B flips a random bit r_1.
2. $[B \to A]$ B sends $c(r_1)$ to A.
3. $[A \to B]$ A sends a random bit r_2 to B.
4. B computes the result of the coin flip $r := r_1 \oplus r_2$.

Of course, B can still assign r arbitrarily, as A has no way of decommitting $c(r_1)$ by herself.

3.1 Malicious Players

Let us return to the problem of protocols which call for B to make secret, but honest, coin flips. Suppose we have reached such a point in a hypothetical exchange. It is B's turn to talk, and the protocol requires him to send some message $f(T, \vec{r})$, where f is a deterministic function of T, the transcript of the conversation recorded so far, and a secret random $\vec{r}$. We will modify this protocol to prohibit B from fixing $\vec{r}$ in his favor.

1. B generates a random string $\vec{r}_1$.
2. B sends $c(\vec{r}_1)$ to A.
3. A sends a random bit $\vec{r}_2$ to B.
4. B computes the result of the coin flip $\vec{r} := \vec{r}_1 \oplus \vec{r}_2$.
5. B sends the message $\alpha := f(T, \vec{r})$.

Now, how can B prove that his message α was according to protocol? That is, he must convince A that $\alpha = f(T, \vec{r}_1 \oplus \vec{r}_2)$, for some $\vec{r}_1$ that is the decommitment of $c(\vec{r}_1)$. Now, f must be a polynomial time algorithm, since B has only polynomially many computing resources. So this statement

$$(\exists \text{ a decommitment scheme } d)[\alpha = f(T, d(c(\vec{r}_1)) \oplus \vec{r}_2)]$$

is an NP-statement. Thus, by NP-completeness it can be reduced to graph 3-colorability, that is, "G is 3-colorable" for some graph G which can be computed in polynomial time and hence both A and B can agree upon. B's proof to A consists of a zero-knowledge proof that G is indeed 3-colorable.

Though this protocol is polynomial-time, it is inefficient. For each message sent by B, A and B exchange extra messages to ensure that B follows the honest protocol. This can be improved on various counts, as we will see next quarter.

3.2 Example: "Poker Over the Phone"

This technique, combined with two-party secure computation, can also be generalized to simulate an objective third party. Suppose a protocol calls for a third party M to output some message $f(T, \vec{r})$ to A but not to B, where $f, T, \vec{r}$ are as before. There are two obstacles here; A and B must jointly generate a random $\vec{r}$ which neither has knowledge of, and they must compute f without letting B know the result. Both problems are easily solved given previous constructions.

1. A generates a random string $\vec{r}^B$ into the well for B.
2. B generates a random string $\vec{r}^A$ into the well for A. $\vec{r}^A$ and $\vec{r}^B$ are A's and B's shares, respectively, to the input $\vec{r}$.
3. A and B compute $(f(T,\vec{r}))^A, (f(T,\vec{r}))^B$ by two-party secure computation.
4. B sends $(f(T,\vec{r}))^B$ to A.
5. A computes M's output $f(T,\vec{r}) = (f(T,\vec{r}))^A \oplus (f(T,\vec{r}))^B$.