# 1 Lecture 13: More OT

## 1.1 Implementing OT

### 1.1.1 Using Trapdoor Permutations

One option is for us to use trapdoor permutations. The sender $S$ and receiver $R$ do the following:

0. $S$ picks 2 bits $b_0, b_1$ and $R$ picks a bit $c$.
1. $S$ sends a trapdoor function publickey, $f$ (but keeps $f^{-1}$ secret)
2. $R$ picks a random $x_c$, and computes $y_c = f(x_c)$
3. $R$ picks a random $y_{\neg c}$
4. $R$ sends back the pair $(y_0, y_1)$
5. $S$ computes $x_0 = f^{-1}y_0$ and $x_1 = f^{-1}y_1$
6. $S$ generates a random $p$, and sends back $p$, $\langle p, x_0\rangle \oplus b_0$, $\langle p, x_1\rangle \oplus b_1$.

Note that a receiver can easily break this by generating both $y_0, y_1$ from randomly selected $x_0, x_1$, instead of randomly selecting one of the $y_c$. So, we have $S$ flip a coin $c$ in to $R$'s well, and $R$ provides a zero-knowledge proof that they are behaving according to the protocol.

Another possible small issue is that $S$ uses a trapdoor function that is not a perfect permutation. If $f$ is not a permutation, it is possible that $R$ is unable to figure out either bit properly.

So, we either need to use a **"certifiable" trapdoor permutation**, which is one where $R$, simply by looking at its code, can verify that it is a trapdoor permutation. Or, we can use a challenge-and-response game by which $R$ can confirm that the permutation is one-to-one on "most" inputs.

### 1.1.2 Using Homomorphic Encryption

For using homomorphic encryption, we assume that it is multiplicative. That is:

$$E(X) \otimes E(Y) = E(X \otimes Y).$$

Then, sender $S$ and receiver $R$ can implement 1-2-OT in the following way:

0. $S$ picks bits $b_0, b_1$ and $R$ picks bit $c$
1. $R$ sends $S$ an encryption key for $E$
2. $R$ generates and sends $y_c = E(\neg c)$ and $y_{\neg c} = E(c)$.
3. $S$ computes and sends $z_0 = E(b_0) \otimes y_c$ and $z_1 = E(b_1) \otimes y_{\neg c}$

Note that as long as $R$ plays honestly (which we can guarantee via ZKP), $R$ will only ever learn one of the $b$'s, since the other one with multiplied by $E(0)$. It is easy to see that $R$ should always get back $b_c$.

Note that the trapdoor permutation implementation hides $b_0, b_1$ computationally, since we are depending on the assumption that the function is hard to invert. And, it hides $c$ information theoretically, since the sender should be able to invert any string of a given length using the trapdoor, so there is no way for them to distinguish which is which.

On the other hand, the homomorphic encryption hides $b_0, b_1$ information-theoretically, since an honest player cannot possibly know what has been multiplied by 0 to give 0. And, $c$ is hidden only computationally, since we are depending on the assumption that encryptions of 0 and 1 are indistinguishable in an encryption scheme that is semantically secure.

As it turns out, it is impossible to information-theoretically hide both if we implement using an encryption scheme.

### 1.2 Beaver's Trick

We note that the following is also called **Random-OT** or **Correlated Randomness**. Beaver's Trick is essentially a way to pre-compute OT. That is, we do the whole communication exchange before the bits $b_0, b_1, c$ have been chosen, so that when they are chosen, we can communicate over any normal channel to complete the 1-2-OT.

This works in two phases: First phase: precomputing the OT

0. $S$ picks random bits $r_0, r_1$ and $R$ picks a random bit $w$
1. $S$ and $R$ perform 1-2-OT with these bits, ending up with:
   (a) $S$ has a pair $(b_0, b_1)$
   (b) $R$ has a pair $(w, r_w)$

These pairs form the correlated randomness. The next phase, after $b_0, b_1, c$ have been chosen, is to finish the 1-2-OT:

1. $R$ computes and sends $z = c \oplus w$.
2. $S$ computes and sends $d_0 = b_z \oplus r_0$ and $d_1 = b_{\neg z} \oplus r_1$.

Note that $c$ is hidden as strongly as $w$, since it is masked by $w$. And, the $b$'s are hidden as strongly as the $r$'s, since they are masked by the $r$'s.

$R$ will ultimately end up with $b_z$ if $w = 0$, or $b_{\neg z}$ if $w = 1$. We can easily see that this means $R$ ends up with $b_c$, as desired.

### 1.3 OT at Bulk (Silent OT)

This is another method of precomputing OT. The basic idea is that some "Silent OT" Machine generates two seeds, one for the sender and one for the receiver, and each party uses the same PRG to generate correlated randomness, which is later used to perform 1-2-OT via the second phase of Beaver's Trick.

### 1.4 1-N-OT and Private Information Retrieval (PIR)

Begin by considering what "1-N-OT" would look like, as an extension of 1-2-OT. The sender would have $n$ pieces of information $x_1, \ldots, x_n$, which might look like a database. The receiver queries for data $x_i$, and via some response from the sender, learns $x_i$. As with 1-2-OT, we want the security guarantee that $S$ must never learn $i$, and $R$ must not learn anything about the other $x_k$.

PIR works by the same mechanism, but we have only the guarantee that $i$ is hdden. Instead of guaranteeing that the receiver learns nothing about the other $x_k$, we focus on making the query and response much smaller than the size of the database ($n$).

#### 1.4.1 PIR from Homo. Encryption

We can implement PIR via additive encryption ($E(X) \otimes E(Y) = E(X \oplus Y)$) in the following way:

1. $S$ breaks teh database into $\sqrt{n}$ blocks $X_k$ of size $\sqrt{n}$.
2. $R$ sends a homomorphic encryption keys to $S$.
3. $R$ sends $y_k = (a_k, b_k)$ for each block, where $a_k = E(0)$ and $b_k = E(1)$ if and only if $x_i$ is in block $X_k$.
4. In each block $X_k$, $R$ replaces all 0's with $a_k$, and all 1's with $b_k$.
5. $S$ computes $A_k = X_1[k] \otimes X_2[k] \otimes \cdots \otimes X_{\sqrt{n}}[k]$ and sends back $A_1, \ldots, A_{\sqrt{n}}$.
6. $R$ decrypts all the $A_k$'s, from which it learns all of $X_k$, including $x_i$

This is private information retrieval because $i$ is computationally hidden by the semantic security of a homomorphic encryption. And, both the query and the answer are of order $O(k\sqrt{n}$, where $k$ is the size of the encryption of a single bit.

Something nice that happens here is that $R$ is forced to behave honestly, in the sense that the best $R$ can do is learn one block, since attempting to learn more than one would mask them over each other and cause the communicated information to be garbage.

In fact, we can do better than $O(\sqrt{n}$. Consider making blocks of size $n^{2/3}$. Then, the receiver would send over a query of size $kn^{1/3}$, which the sender would use to mask the blocks over one another and end up with $n^{2/3}$ total resulting answers. From here, the sender has a database of size $n^{2/3}$, and we are able to do the same $\sqrt{n}$ protocol as above to finish it off in $O(n^{1/3})$.

### 1.4.2 $\implies$ Collision-Resistant Hash Functions

Suppose that we have a functioning private information retreival system. The querying system, which returns some homomorphically encrypted answer for any given index $i$ is a length-decreasing function on the entire database. And, we claim that no **PPT** adversary can find any two distinct databases such that the answer to a query for some random $i$ looks the same for both databases.

Assume otherwise. If the two databases are different, then two entries must differ in at least one spot, so the private information retrieval cannot possibly give the same response from both databases if the query requests that index. So, if the answers to a request for $i$ in each database are the same (i.e., we have a collision), then this leaks the information that we did not request anything in the block where the two databases differ. However, we may not leak any information about $i$ during private information retrieval, so this breaks its security, and creates a contradiction.

### 1.4.3 *implies* 1-N-OT

As it turns out, PIR is actually stronger than 1-N-OT, even though we do not make the guarantee in general that the receiver does not learn anything about other entries.

For a database of length $n$, the indices are length $\log(n)$. We generate $2\log(n)$ PRF keys in pairs, where $(s_0, s_1)_j$ is the pair corresponding to bit $j$ of the index. Then, we mask each entry in the database (via a method of our choice) using the PRF's corresponding to the bit values of the index of the entry. The receiver can then perform PIR to get a masked block of the database, then performs 1-2-OT $log(n)$ times to get the keys corresponding index. The receiver can then use these keys to figure out only the desired entry in the database.