1 Lecture 12: Oblivious Transfer (OT)

1.1 Basics of OT

1.1.1 Rabin-OT

The idea of Rabin OT is the following process:

1. A sender S sends a bit b
2. b goes through an Oblivious Transfer channel
3. A receiver R receives something:
   (a) With 50% probability, R receives b
   (b) With 50% probability, R receives garbage, which we denote as #.

Additionally, we have two privacy guarantees here. S should never be able to figure out whether R got b or #. R should never be able to figure out what b was, if they received #.

1.1.2 One-out-of-two OT (1-2-OT)

This is similar to Rabin OT:

1. S sends two bits b_0 and b_1.
2. the bits go through an oblivious transfer channel.
3. R provides a bit c to the channel.
4. R receives the bit b_c.

Similarly, we have two privacy guarantees. S should never be able to figure out which bit R requested and received. R should never be able to figure out anything about the bit $b_{\neg c}$.

1.1.3 Equivalence of Rabin-OT and 1-2-OT

It is easy to see that 1-2-OT can be used to create Rabin-OT.

We make one "clear" channel, over which the sender guarantees that both bits will always be the same, correct bit. We also make one channel which will be used for the OT. Over the OT channel, S will randomly choose one of b_0 or b_1 to be the correct information, and will choose the other to be "garbage" (a random bit that doesn't mean anything). Then, with 50/50 probability, R will choose a random bit and will get the correct information or the garbage. We finish by having S send a bit indicating which one had the correct information, over the clear channel. This allows R to know whether it got the correct bit or the garbage.

It's not hard to see that these properties match up with Rabin-OT.

The more difficult direction was proven by Claude Crepeau, and involves some creative construction.

1. R sends $3k$ bits over a Rabin-OT channel
2. S either receives bit b_i or # for each of the $3k$ bits.
   (a) R should expect to receive $3k/2$ bits (i.e., not garbage)
   (b) R has a negligible probability of receiving more than $2k$ bits or fewer than k bits
3. R chooses two **disjoint, size-k** subsets $I_0, I_1 \subset \{1, 2, \ldots, 3k\}$.
   (a) Ignoring the negligible probability cases mentioned above, the best R can do is know all the bits corresponding to exactly one of these sets
4. R sends the bit-encodings of I_0, I_1 (i.e., a $3k$-bit string with $b_i = 1$ i.f.f. $i \in I_c$)
5. S checks that $|I_0| = |I_1| = k$ and $I_0 \cap I_1 = \emptyset$.
6. S sends back $b_0 \oplus \langle I_0, b_1 b_2 \cdots b_{3k} \rangle$ and $b_1 \oplus \langle I_1, b_1 b_2 \cdots b_{3k} \rangle$.
   (a) R will only be able to figure out b_c if they know all the b_i's corresponding to I_c. So except for the negligible probability case where R receives k or fewer bits, or $2k$ or more bits, R will get exactly one of b_0 or b_1.

Note that since Rabin-OT guarantees that S cannot know when the bit goes through or garbage goes through, S cannot know which set out of I_0 or I_1 is fully known by R, so S cannot know which bit out of b_0 or b_1 is known by R at the end.

So this satisfies all the constraints of 1-2-OT.

1.1.4 Computing AND via 1-2-OT

Note that we may compute AND via 1-2-OT, with the expected restriction that if one player's input to the AND is 1, they will certainly be able to figure out the other player's input simply due to the nature of AND.

However, we do get the guarantee that if a player's input is 0, they will not be able to learn anything about the other player's input.

It works simply as follows:

1. S inputs $b_0 = x \cdot 0 (= 0)$ and $b_1 = x \cdot 1 (= x)$
2. R inputs y
3. R gets out $b_y = x \cdot y$.

1.2 2-Party Secure Computation

The more interesting generalization of the small example we did above is multi-party computation. As it turns out, we can implement a special case of this using 1-2-OT. The basic building block we will use is called **Additive Secret Sharing**.

1.2.1 Additive Secret Sharing

In general, secret sharing is where we split up a certain piece of information among different parties so that unless all parties present the information that they have, they will not be able to figure out the original piece of information.

In additive secret sharing, we split it up amongst different parties (in this case, 2 parties) so that when they add up their individual secrets modulo some prime p, they will end up with the original secret. It is easy to see that this works since, without any one of them there, there's no way of knowing what the last number is, so the possibilities of the original secret remain evenly distributed over $\mathbb{Z}/p\mathbb{Z}$.

1.2.2 Building Secure 2-Party Computation with 1-2-OT

To begin, 2-Party Computation is a special case of Multi-Party Computation (MPC) where only 2 parties are involved. In general, MPC is where several different parties combine their data to perform some computation, but they do it in such a way that no party learns anything about the other parties' data, except for information that would be implied by the output and their own data.

We begin by noting that modular multiplication ($\otimes$) and addition ($\oplus$) (specifically, mod 2) form a universal set of gates. Note that these operations are precisely bitwise AND and bitwise XOR. We can construct a NOT gate by XOR'ing with 1, and w can construct an OR gate as: $(x \oplus y) \oplus (x \otimes y)$. We know that AND, NOT, and OR form a universal set, so this shows that $\oplus, \otimes$ form a universal set as well.

So the first step in our computation is to convert f into a digital circuit composed only of $\otimes$ and $\oplus$ gates.

Then, we employ secret sharing. We begin by having each party secret-share each of their inputs with the other party. For example, if x is an input of party A, then A finds $x_1 \oplus x_2 = x$ and gives x_2 to B.

If, given two secret-shared wires as input to a gate, we can find a way to generate the proper output as a secret-shared wire without breaking the secret-sharing on the input, then we can complete the whole computation and secret share the final output. Then, the two parties can combine their secrets to figure out the output, but they won't reveal anything that happened before– specifically, their inputs.

So, all we need to do is figure out how, to get a secret sharing of $(a \oplus b) \oplus (c \oplus d)$, and $(a \oplus b) \otimes (c \oplus d)$, where one party knows a and c, and the other party knows b and d.

For addition, this is easy. The new secret sharing would simply be $(a \oplus c) \oplus (b \oplus d)$, since modular addition is associative and commutative.

For multiplication, it is a bit more complicated. When we expand out the expression, we have

$$(a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c) \oplus (b \otimes d).$$

So the first party already has the first term, and the second party already has the last term. But we run into an issue with the center terms, since giving them to either party could potentially break the secret sharing of the input wires. An initial idea might be to use 1-2-OT to compute the AND of a and d, but this gives away a.

So we just make a small tweak to fix this. Instead of computing $a \otimes d$ with the 1-2-OT, we compute $(a \otimes d) \oplus r$, where r is some random bit. Then, what we end up with is that the first party has r, and the second party has $(a \otimes d) \oplus r$. We do this similarly for $b \otimes c$, so that the first part has a random bit s and the second party has $(b \otimes c) \oplus s)$.

The first party doesn't know the second party's secrets due to the properties of 1-2-OT, and the second party doesn't know the first party's secrets due to the addition of the random bit. We end up with the final secret sharing of the output:

$$[(a \otimes d) \oplus r \oplus s] \oplus [((a \otimes d) \oplus r) \oplus ((b \otimes c) \oplus s) \oplus (b \otimes d)] .$$

So, the two parties can safely implement 2-party computation by doing this for one gate at a time, and then sharing their secrets on the final output