# CSE 5854: Class 11

Benjamin Fuller

February 22, 2018

## 1 Working with oblivious transfer

In this class we are going to show some of the flexibility of oblivious transfer. First we'll show that once we have the ability to transfer one out of two strings we gain flexibility on the number of possible strings. This is known as 1 out of $n$ oblivious transfer. We won't show it in class but this also allows oblivious transfer where the receiver gets multiple strings. This is known as $k$ out of $n$ oblivious transfer.

The first oblivious transfer construction we saw (using oblivious sampling of a public key encryption) is very easy to increase the number of possible choices. The receiver just samples more public keys where the private key is unknown. However, we may want flexibility in protocol choices or require an oblivious transfer protocol that is maliciously secure. Thus, we will try and build a 1 out of $n$ oblivious transfer using a 1 out of 2 oblivious transfer as a black box. That is, we can provide inputs and outputs but don't know the internals of how it works.

The idea behind the construct is to combine 1 out of 2 oblivious transfer with a binary tree of encryptions. We will consider the case of 1 out of 4 oblivious transfer but the idea naturally generalizes to a larger number of strings.

The idea is to use symmetric encryption to encrypt the possible values $x_0, x_1, x_2, x_3$. Then rather than initializing the 1-out-of-2 oblivious transfer on the values $x_i$, we initialize them with keys to the encryptions. The main detail of the protocol is how to set up the keys so that the receiver can only decrypt one of these values (and it is possible for them to specify all four choices). So we think of four keys specified by the sender $r_0, r_1$ and $r_{*0}, r_{*1}$. The keys $r_i$ notionally represent the high order bit of $d$ and the keys $r_{*i}$ represent the low order bit of $d$. Before running any OT the sender prepares the following 4 encryptions. These encryptions are sent to $R$ in a random order.[1]

| String value | Key 1 | Key 2 | Ciphertext |
|---|---|---|---|
| $x_0$ | $r_0$ | $r_{*0}$ | $\mathsf{Enc}_{r_0 \oplus r_{*0}}(x_0)$ |
| $x_1$ | $r_0$ | $r_{*1}$ | $\mathsf{Enc}_{r_0 \oplus r_{*1}}(x_1)$ |
| $x_2$ | $r_1$ | $r_{*0}$ | $\mathsf{Enc}_{r_1 \oplus r_{*0}}(x_2)$ |
| $x_3$ | $r_1$ | $r_{*1}$ | $\mathsf{Enc}_{r_1 \oplus r_{*1}}(x_3)$ |

Then the sender and receiver setup two 1-out-of-2 oblivious transfers. The first $S$ initializes with $r_0$ and $r_1$ as the possible inputs and the second with $r_{*0}$ and $r_{*1}$ as possible inputs.

[1] It is also possible to think of sending double encryptions rather than an encryption with the $\oplus$ of the two keys. The analysis follows similarly.

To summarize the protocol works as follows (from the senders point of view):

1. Generate four random keys,
2. Encrypt input values under these keys,
3. Engage in OT with the receiver so that one encrypt is decryptable.

**Efficiency:** The first thing to note is that the complexity of this protocol grows linearly with the number of possible values to be transferred plus a logarithm number of oblivious transfers.

**Security:** If the underlying oblivious transfer is maliciously secure it should be clear that the receiver has no opportunity to cheat here. They have to specify two input bits and they will be able to decrypt at most one value. It is easy to see that the other three keys are hard to distinguish from random (assuming a sufficiently strong symmetric encrypt). However, a malicious $S^*$ has more of an opportunity to cheat as they are preparing the encryptions and initializing the oblivious transfer with values that are supposed to be connected to the prepared encryption. Here are some ways that $S^*$ could try and cheat:

1. They could make 1 or more of the encryptions malformed. This might be done in the hope of having $R$ complain at the end of protocol and using this to detect which value was used by $R$.
2. They could fail to initialize the 1-out-of-2 oblivious transfers with a value that is not connected to the keys used to produce the encryptions.
3. They could initialized the oblivious transfers with connected keys.
4. They could encrypt multiple values under the same oblivious transfer.

We ignore problems 3 and 4 as these only help the receiver in the sense that they may be able to decrypt multiple values. However, problems 1 and 2 could be serious as they cause the protocol to fail in a way that depends on the value of $R$. If $R$ chooses to silently fail this won't be detectible by $S^*$ but it allows a kind of conditional transference of values. On the other hand if $S$ complains this tells $S^*$ the value of $R$. Truly preventing this problem requires a full malicious transform using zero knowledge proofs of knowledge (ZKPoK). We will discuss this remediation next. However, this remediation only makes it difficult for $S^*$ to cause selective abort in a way that depends on $R$'s value. It does not prevent $S^*$ from causing the protocol to fail.

# 2 Semi-honest to malicious OT

Last class we described a semi-honest oblivious transfer protocol. Today we will show how to make it maliciously secure.

We presented a semi-honest oblivious transfer as follows:

| $S$ | $R$ |
|---|---|
| 1. Input $x_0, x_1$. | |
| | 1. Input $b$. |
| | 2. Sample $pk_b, sk_b \leftarrow \mathsf{Gen}(1^k)$. |
| | 3. Sample $pk_{1-b} \leftarrow \mathsf{Samp}(1^k)$. |
| 5. Receive $pk_0, pk_1$. | 4. Send $pk_0, pk_1$. |
| 6. Encrypt $c_0 = \mathsf{Enc}_{pk_0}(x_0), c_1 = \mathsf{Enc}_{pk_1}(x_1)$. Send to $R$. | |
| | 7. Receive $c_0, c_1$, decrypt $c_b$. |

We note that the sender can cheat in the same way as the previous protocol: it can cause $R$ to fail only in some cases (that depends on $R$'s bit). However, for the moment we will focus on providing security against a malicious receiver $R^*$. Here $R^*$ can cheat in a very obvious way instead of running $\mathsf{Samp}$ they can run the $\mathsf{Gen}$ algorithm and have access to both public keys. This will be the first example of a general protocol for enforcing honest behavior. There are two main components to the protocol. First we want to enforce that $R^*$ actually uses the algorithm $\mathsf{Samp}$ to produce on of the public keys. This can be enforced by using a zero-knowledge proof of knowledge for the following language. Let pairs $pk_0, pk_1$ be possible elements of $L$ defined as the set

$$L = \{pk_0, pk_1 | \exists r \text{ such that } pk_0 = \mathsf{Samp}(r) \text{ or } \exists r \text{ such that } pk_1 = \mathsf{Samp}(r)\}.$$

If we force the receiver to initiate such a proof it ensures that they have run $\mathsf{Samp}$.[2] However, it does not ensure that they don't know the corresponding secret key. To do this we have to take the choice of $r$ out of the hands of $R^*$. To do this we initiate a coin flipping protocol and make $R^*$ prove consistency with this protocol.

[2]Security of this protocol depends on the $\mathsf{Samp}$ protocol. If $\mathsf{Samp}$ generates both keys and then just throws out the secret key no modification can possibly make this secret. However, for many algorithms such as El Gamal sampling a public key without knowing the public key is easy, it is just sampling a random group element.

| $S$ | $R$ |
|---|---|
| 1. Initiate coin flipping with transcript $\tau$. | 1. Initiate coin flipping protocol with transcript $\tau$. Receive $r = r_0 \| r_1$ |
| | 2. Sample $pk_b, sk_b \leftarrow \mathsf{Gen}(r_0)$. |
| | 3. Sample $pk_{1-b} \leftarrow \mathsf{Samp}(r_1)$. |
| | 4. Send $pk_0, pk_1$. |
| 6. Receive $pk_0, pk_1$, verify ZKPoK. | 5. Prove that $pk_b$ was produced by running $\mathsf{Gen}$ on $r_0$ and $pk_{1-b}$ by running $\mathsf{Samp}$ on $r_1$ which are the output of the coin flipping protocol with transcript $\tau$. |
| 7. Encrypt $c_0 = \mathsf{Enc}_{pk_0}(x_0), c_1 = \mathsf{Enc}_{pk_1}(x_1)$. Send to $R$. | |
| | 8. Receive $c_0, c_1$, decrypt $c_b$. |

In essence we are forcing $R^*$ to run the honest $R$ on some random coins that were jointly decided by both parties. The only freedom $R^*$ has here is to select input $b$ in a way that depends on the coin flipping output. We can take away this freedom by forcing $R^*$ to commit to its input before the coin flipping protocol. The final protocol looks like this:

| $S$ | $R$ |
|---|---|
| | 1. Commit to $c = \mathsf{Commit}(b)$ to $R$. |
| 2. Receive $c$ | |
| 3. Initiate coin flipping with transcript $\tau$. | 3. Initiate coin flipping protocol with transcript $\tau$. Receive $r = r_0 \| r_1$ |
| | 4. Sample $pk_b, sk_b \leftarrow \mathsf{Gen}(r_0)$. |
| | 5. Sample $pk_{1-b} \leftarrow \mathsf{Samp}(r_1)$. |
| | 6. Send $pk_0, pk_1$. |
| 6. Receive $pk_0, pk_1$, verify ZKPoK. | 7. Prove that $pk_b$ was produced by running $\mathsf{Gen}$ on $r_0$ and $pk_{1-b}$ by running $\mathsf{Samp}$ on $r_1$ which are the output of the coin flipping protocol with transcript $\tau$ where $b$ is value committed to in $c$. |
| 7. Encrypt $c_0 = \mathsf{Enc}_{pk_0}(x_0), c_1 = \mathsf{Enc}_{pk_1}(x_1)$. Send to $R$. | |
| | 8. Receive $c_0, c_1$, decrypt $c_b$. |

The reader should convince themselves that the language whose membership is being proved by $R^*$ is in fact NP. Roughly, this is because it is arguing about the output of polynomial time algorithms when using a particular input.

Furthermore, its worth noting the round complexity of the revised protocol. Before we had a two message oblivious transfer (keys from $R$ to $S$ and ciphertexts back. Now we:

1. A commitment message from $R$ to $S$.
2. A two message coin flipping protocol.
3. The public keys augmented with a zero knowledge proof of knowledge. Recall that a ZKPoK with good soundness and extraction requires five rounds of interaction.
4. The two ciphertexts from $S$.

With some scheduling these messages can be condensed into 8 total messages (and recall that the length of each message ZKPoK scales with soundness). In the general case we need to do this with each message sent by a potentially malicious party. This result should be seen as a feasibility result. It is possible to enforce honest behavior while maintaining polynomial time but this protocol is very slow and more efficient algorithms exist.