University of Illinois, Urbana Champaign
CS/ECE 498AC3/4 Applied Cryptography

Instructor: Dakshita Khurana
Scribe: Amit Agarwal
Date: November 05, 2020

LECTURE
21

Two Party Computation 1: Secret Sharing and Computations

In the last lecture, we looked at Oblivious Transfer (OT) functionality and a candidate protocol for OT in the Random Oracle Model (ROM). We also briefly discussed the security proof of the OT protocol against a malicious sender and receiver. In this lecture, we will revisit the security proof for the OT protocol in detail. Then we will see how, using OT as black-box, one can construct a protocol for securely computing any arbitrary function (such as AND, OR, NAND etc).

21.1 Background

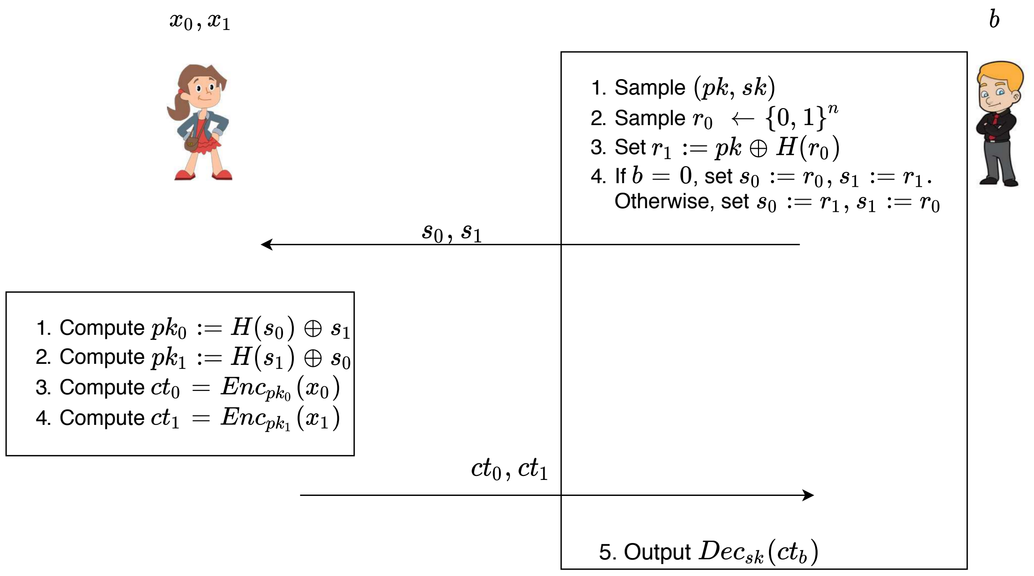
Recall that the aim of an OT protocol is to perform the following task: A sender S has inputs x_0, x_1 whereas the receiver R has a choice bit b. At the end of protocol, R should receive x_b. The informal security requirement is the following: i) A malicious S should not learn b, ii) A malicious R should not learn x_{1-b}. To formalize the security requirement, we resort to the Real-Ideal paradigm where we compare the execution of an ideal-world experiment of OT to its real-world counterpart.

In the ideal world, we assume that there is a trusted party which computes the desired functionality (e.g. OT) by receiving the inputs from all parties and sending them back the respective outputs. In the real world, there is no trusted party, and all parties are supposed to run the prescribed protocol for computing the functionality. The issue is that, in the real-world, there might be inadvertent leakage of information about private inputs of parties. Also, malicious parties can chose not to follow the prescribed protocol in the real-world. Hence, to quantify the security of a real-world protocol, we compare its execution to the ideal-world experiment which, by definition, is secure. In the last lecture, we formalized the security definition in the Real-Ideal paradigm using a simulator Sim whose task is to interact with the corrupt party in the ideal world and create a view that is indistinguishable from the real world view of the adversary [1].

[1]Actually, the stronger (and necessary) security requirement is that the joint distribution of simulated view of the adversary along with honest party's output in the ideal world should be computationally indistinguishable from the joint distribution of real-world view of the adversary and real world honest party's

21.2 A maliciously secure OT protocol

Last time, we looked at a candidate OT protocol in the ROM model and this is shown in Fig 21.1. We claimed the security of this OT protocol against malicious senders and receivers. Security against a malicious sender guarantees that even a sender who deviates from the protocol in an arbitrary fashion will not gain any information about receiver's choice bit b. Similarly, security against a malicious receiver implies that a receiver who arbitrarily deviates from the prescribed protocol will not succeed in learning both the sender's messages x_0, x_1. The high-level intuition as to why the protocol satisfies such strong notion of security was the following: The use of Random Oracle ensures that there is atleast one public key, out of pk_0 and pk_1, for which a malicious receiver does not know the corresponding secret key and therefore will not be able to recover both the messages due to CPA security of public-key encryption scheme. With respect to a malicious sender, we argued that the receiver's choice bit would remain hidden because s_0, s_1 are indistinguishable from uniformly random values. [2]



x_0, x_1
b
1. Sample (pk, sk)
2. Sample $r_0 \leftarrow \{0,1\}^n$
3. Set $r_1 := pk \oplus H(r_0)$
4. If $b = 0$, set $s_0 := r_0$, $s_1 := r_1$. Otherwise, set $s_0 := r_1$, $s_1 := r_0$
s_0, s_1
1. Compute $pk_0 := H(s_0) \oplus s_1$
2. Compute $pk_1 := H(s_1) \oplus s_0$
3. Compute $ct_0 = Enc_{pk_0}(x_0)$
4. Compute $ct_1 = Enc_{pk_1}(x_1)$
ct_0, ct_1
5. Output $Dec_{sk}(ct_b)$

FIGURE 21.1: Maliciously-secure OT protocol in the ROM model

21.2.1 Security against malicious sender

To formally prove security against a malicious sender, we need to come up with a simulator Sim_S which can simulate the real-world view of a malicious sender in the ideal world as

output. However, since OT is a deterministic functionality, we can safely ignore the honest party's output for now. More details regarding this can be found in Section 4 of [1]

[2]Along the way, we also made a simplifying assumption that one can sample pseudo-random public keys from the Gen algorithm of the public-key encryption.

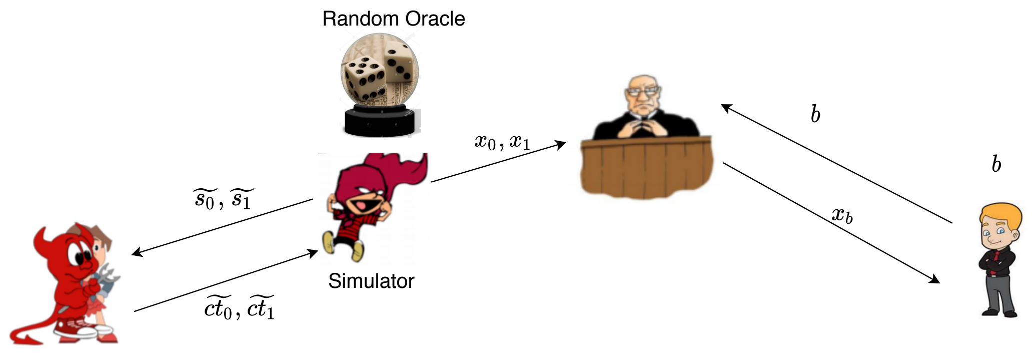

Random Oracle
x_0, x_1
b
b
$\widetilde{s_0}, \widetilde{s_1}$
x_b
Simulator
$\widetilde{ct_0}, \widetilde{ct_1}$

Figure 21.2: Simulator for a malicious sender in the OT functionality in ROM model

illustrated in Fig 21.2. Note that this involves *extracting* the messages x_0, x_1 from the sender's protocol messages and then sending both of them to the trusted party. To do so, the simulator can use the "programmability" power of the RO i.e. it can program the outputs of the RO in a way so that $\mathsf{Sim_S}$ knows the secret-keys underlying both pk_0 and pk_1. Once this is done, the remaining part is relatively straightforward: $\mathsf{Sim_S}$ on receiving $\widetilde{ct_0}, \widetilde{ct_1}$ can just decrypt them using sk_0, sk_1 respectively to obtain x_0, x_1. $\mathsf{Sim_S}$ can then simply send x_0 and x_1 to the trusted party. We will now discuss the steps taken by $\mathsf{Sim_S}$:

1. Generate (pk, sk) and (pk', sk') using the Gen algorithm for PKE.
2. Randomly sample $\widetilde{s_0}, \widetilde{s_1}$
3. Set $H(s_0) := pk \oplus \widetilde{s_1}$
4. Set $H(s_1) := pk' \oplus \widetilde{s_0}$
5. Send $\widetilde{s_0}, \widetilde{s_1}$ and receive $\widetilde{ct_0}, \widetilde{ct_1}$
6. Compute $x_0 := \mathsf{Dec}(sk, ct_0)$ and $x_1 := \mathsf{Dec}(sk', ct_1)$
7. Send (x_0, x_1) to the trusted party.

We will briefly analyze the simulation strategy that we sketched above. Note that in the real world, S receives two messages s_0, s_1 from R, both of which are uniformly random. Therefore, we need to make sure that our $\mathsf{Sim_S}$ is also able to match the same distribution. Indeed, as mentioned in Step 2, the $\widetilde{s_0}, \widetilde{s_1}$ are sampled randomly and therefore would be indistinguishable from s_0, s_1. Also, note that the particular way in which we programmed the RO (in Steps 3 and 4) ensures that the public-keys pk_0, pk_1 that the sender computes (in Step 1 and 2 of Fig 21.1) indeed matches the sampled public-keys pk, pk' (in Step 1 of the simulation) respectively, and $\mathsf{Sim_S}$ indeed knows the secret-keys underlying both these public-keys simply because it sampled them by itself. Therefore, after receiving the messages x_0, x_1 from the S in Step 6, $\mathsf{Sim_S}$ simply decrypts them using sk, sk' respectively, and then send the messages to the trusted party.[3]

[3]Note that a malicious sender could infact choose not to encrypt his inputs x_0, x_1 under pk_0, pk_1 respectively but this is not a problem because in that case even a real world receiver would not be able to recover the correct message. Therefore, $\mathsf{Sim_S}$ can safely send whatever the decryption output is (even junk) to the trusted party.

21.2.2 Security against malicious receiver

To formally prove security against a malicious receiver, we need to come up with a simulator Sim_R which can simulate the real-world view of a malicious receiver in the ideal world as illustrated in Fig 21.3. Note that this involves *extracting* the choice bit b from the receiver's protocol messages and then sending b to the trusted party to retrieve the message x_b. The challenge is that Sim_R should be able to perform this just by looking at $\{s_0, s_1\}$ since those are the only protocol messages sent by the receiver in the protocol. At this point, one might wonder that if there was really a way to extract bit b by looking at $\{s_0, s_1\}$, then why can't a sender, in the real world, also compute b and thus break receiver's privacy? The answer is that the simulator will have an edge over the real-world prover: It can observe and program all the queries that the receiver makes to the Random Oracle RO. Now we will describe the exact procedure that Sim_R will employ in order to extract choice bit b, and once this is done, the rest of simulation will be relatively straightforward.

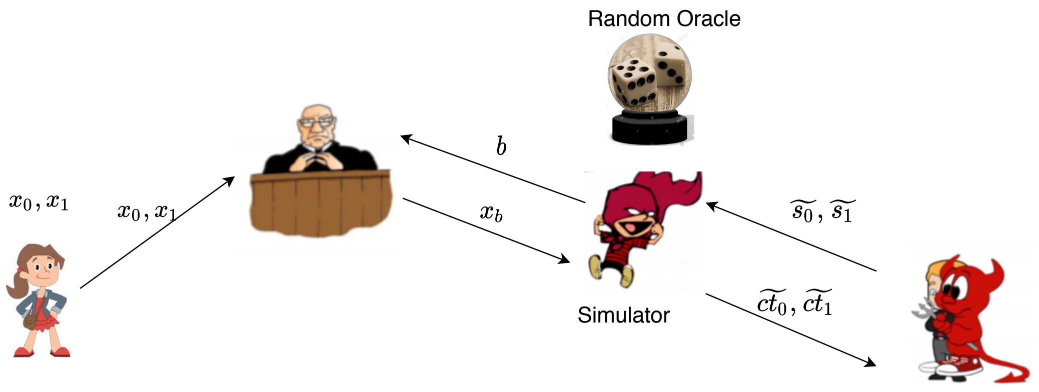

Random Oracle
x_0, x_1
x_0, x_1
b
x_b
$\widetilde{s_0}, \widetilde{s_1}$
$\widetilde{ct_0}, \widetilde{ct_1}$
Simulator

Figure 21.3: Simulator for a malicious receiver in the OT functionality in ROM model

We start by pointing out that the following fact: For any value v, a malicious receiver cannot know the output of RO on v i.e. $H(v)$ without actively querying the RO. This is known as "observability" property which means that Sim_R can *observe* all the queries that the adversary makes to the RO. Having said that, we will split our security analysis into 2 cases: i) R does query the RO for atleast one of the values in $\{\widetilde{s_0}, \widetilde{s_1}\}$, ii) R does not query the RO on either of $\{\widetilde{s_0}, \widetilde{s_1}\}$. The latter case will be relatively easier to handle compared to the former one.

To handle the former case, we will build up on the idea we discussed in the last lecture: namely that Sim_R will try to find out which, out of $\widetilde{s_0}, \widetilde{s_1}$, was queried first to the RO. The intuitive reason why this idea helps is as follows: Note that in the honest protocol, the receiver is supposed to query the RO at one of the two values - either s_0 or s_1 - and then derive the other value by xoring the pk with the output of RO on the queried point. In other words, if s_v was queried first, then $s_{1-v} = pk \oplus H(s_v)$. This means that even a malicious receiver only has a single degree of freedom i.e. for any pk that it has in its mind, it can only fix either s_0 or s_1 arbitrarily but not both. Since the other public key pk' is derived by doing $H(s_{1-v}) \oplus s_v$, and s_{1-v} was outside the control of R, this means that pk' would be a uniformly random value for which R will not know the corresponding sk'. With this idea in place, we can now describe the simulator's algorithm:

$\mathsf{Sim_R}$:

1. Maintain an ordered list L of all the RO queries made by R
2. On receiving $\widetilde{s_0}, \widetilde{s_1}$, extract the underlying choice bit b in the following way:
   - Case 1 - $\widetilde{s_0} \notin L$ and $\widetilde{s_1} \notin L$: In this case, R does not query RO on either $\widetilde{s_0}$ or $\widetilde{s_1}$. Here we can argue that R will not know the sk for any of the public keys pk_0 and pk_1 that the sender S encrypts her messages under. This is because, in the honest protocol, S will compute $pk_0 := H(\widetilde{s_0}) \oplus \widetilde{s_1}$ and $pk_1 := H(\widetilde{s_1}) \oplus \widetilde{s_0}$. Since R never queries the RO on either $\widetilde{s_0}$ or $\widetilde{s_1}$, therefore both pk_0 and pk_1 will just be uniformly random public keys for which R will not know either sk_0 or sk_1, and hence will not be able to decrypt either $\widetilde{ct_0}$ or $\widetilde{ct_1}$ in the Real world. Hence, $\mathsf{Sim_R}$ can just set the choice bit b to be arbitrarily either 0 or 1 and be confident that no matter what message it receives back from the trusted party (i.e. x_0 or x_1), the ciphertexts $\widetilde{ct_0}$ and $\widetilde{ct_1}$ that it forms will not not reveal anything to R.
   - Case 2 - $\widetilde{s_0} \in L$ and $\widetilde{s_1} \notin L$: In this case, R queries RO on $\widetilde{s_0}$ but not $\widetilde{s_1}$. Here we claim that R potentially knows the sk underlying pk_0. This is because, in the Real world, an honest R with choice bit $b = 0$ would have queried RO on s_0 first and derived $s_1 := pk \oplus H(s_0)$. Therefore, $\mathsf{Sim_R}$ also needs to send $b = 0$ in order to retrieve the correct message x_0 from the trusted party.
   - Case 3 - $\widetilde{s_0} \notin L$ and $\widetilde{s_1} \in L$: This is similar to Case 2 except that the R queries RO on $\widetilde{s_1}$ instead of $\widetilde{s_0}$. Here we can claim that R potentially knows the sk underlying pk_1. This is because, in the Real world, an honest R with choice bit $b = 1$ would have queried RO on s_1 first and derived $s_0 := pk \oplus H(s_1)$. Therefore, $\mathsf{Sim_R}$ also needs to send $b = 1$ in order to retrieve the correct message x_1 from the trusted party.
   - Case 4 - $\widetilde{s_0} \in L$ and $\widetilde{s_1} \in L$: This is a tricky case because the malicious R queries RO on both $\widetilde{s_0} \in L$ and $\widetilde{s_1}$, and yet $\mathsf{Sim_R}$ needs to correctly determine which public-key, out of pk_0 and pk_1, does R potentially know the corresponding secret-key for. Here, we can use the idea that we discussed in an earlier paragraph and check which value, out of $\widetilde{s_0}$ and $\widetilde{s_1}$, was the *first* one to be queried to RO. The $\mathsf{Sim_R}$ can easily do this because it observes all the queries made by R to the RO and maintains an ordered list L of all such queries.

     If $\widetilde{s_0}$ was queried first, then R potentially knows the sk underlying pk_0 and we can therefore reduce this to Case 2, and let $\mathsf{Sim_R}$ set $b := 0$. On the other hand, if $\widetilde{s_1}$ was queried first, then R potentially knows the sk underlying pk_1 and we can therefore reduce this to Case 3, and let $\mathsf{Sim_R}$ set $b := 1$.
3. Send the extracted bit b to the trusted party and receive x_b in return
4. Compute the public keys pk_0 and pk_1:

$$pk_0 := H(\widetilde{s_0}) \oplus \widetilde{s_1}$$

$$pk_1 := H(\widetilde{s_1}) \oplus \widetilde{s_0}$$

5. Compute the ciphertexts $\widetilde{ct_0}$ and $\widetilde{ct_1}$:

$$\widetilde{ct_b} = \mathsf{Enc}(pk_b, x_b)$$
$$\widetilde{ct_{1-b}} = \mathsf{Enc}(pk_{1-b}, 0^n)$$

6. Send $\widetilde{ct_0}$ and $\widetilde{ct_1}$ to R

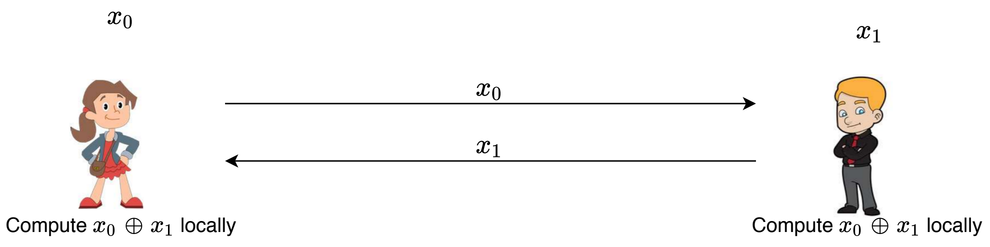
Now we need to additionally prove that the simulated view constructed by $\mathsf{Sim_R}$ in the Ideal world is indistinguishable from the actual view of a malicious R in the Real world. This can be shown using a reduction to the CPA security of the underlying Public-Key Encryption scheme PKE. We will briefly discuss the high-level idea here: Suppose there exists a malicious receiver R which can distinguish between the real and simulated view, then we can construct a wrapper adversary Adv around R which will break the CPA security of the PKE scheme. Adv would interact with a PKE challenger and receive encryptions of two messages m_0, m_1 of his choice. Suppose Adv receives $\mathsf{Enc}(m_t)$, then the task of Adv is to correctly guess the bit t. To do so, Adv would internally interact in the OT protocol as a sender S with the malicious R, and set the ciphertexts $\widetilde{ct_0}, \widetilde{ct_1}$ in a way so that $\widetilde{ct_{1-b}} = \mathsf{Enc}(m_t)$. Adv can do so because, similar to the $\mathsf{Sim_R}$, it can observe all queries made by R to the RO and find the choice bit b. Once Adv is successful in planting $\widetilde{ct_{1-b}} = \mathsf{Enc}(m_t)$, it can forward the distinguishing guess bit g of the malicious R to the outside PKE challenger, and thus contradict the CPA security of the PKE scheme. Such a reduction strategy ensures that any distinguishing advantage of malicious R between the Real and Ideal OT view (for e.g. due to some leakage about the message $x_{1-b} = m_t$) is directly translated into the distinguishing advantage of the wrapper adversary in breaking the CPA security of PKE scheme.

21.3 Universality of Oblivious Transfer

Now we will see the usefulness of OT in constructing secure multi-party computation protocols. In a nutshell, just like how using NAND gates one can contruct a circuit for computing any boolean function f, similarly, using OT one can contruct a *secure protocol* for computing any boolean function f which takes its inputs from multiple parties. In a sense, OT is a *universal* primitive in the context of secure computation.

Suppose Alice (with input bit x_0) and Bob (with input bit x_1) would like to compute $f(x_0, x_1)$. We will begin by looking at ways to perform secure computation for some simple functions. As a side note, we will only focus towards *semi-honest* adversaries in the following constructions. Security against malicious parties can be obtained by using maliciously-secure OT along with commitment schemes and zero-knowledge proofs to enforce honest behavior. However, we will not go into the details of how this is done as it is outside the scope of this lecture.

- **Secure XOR**: In this case, Alice and Bob would like to compute $x_0 \oplus x_1$ in a secure fashion i.e. without revealing their inputs. However, note that the output of XOR function, combined with one of the inputs, directly reveals the other input. For example, Alice on getting the output $c = x_0 \oplus x_1$, can perform $c \oplus x_0$ to retrieve Bob's input x_1. Similarly, Bob can retrieve Alice's private input x_0. Hence, there is trivial protocol for securely computing the XOR function: Each party just sends over their private input to the other party and then each party can locally compute the XOR function on both inputs and output it. This is illustrated in Fig 21.4. Another way to look at it is the following: Since the output of *ideal* XOR functionality itself reveals the private input of parties, therefore leakage of private inputs in the real world protocol is not really a security issue.



x_0
x_1
x_0
x_1
Compute $x_0 \oplus x_1$ locally
Compute $x_0 \oplus x_1$ locally

FIGURE 21.4: Protocol for securely computing the XOR function

- **Secure AND**: Unlike the XOR functionality, AND is non-trivial because the output doesn't necessarily leak the input of parties. For example, consider a case where $x_0 = 1$ and $x_1 = 0$. Therefore, both Alice and Bob will learn the output $x_0 \wedge x_1 = 0$. Although Alice can infer from the output that Bob's input x_1 must have been 0, however, Bob cannot infer from the output whether Alice's input x_0 was 0 or 1. Therefore, since the ideal functionality doesn't necessarily leak the private inputs of parties, we need to make sure that our real-world protocol also maintains the same hiding guarantee.

  Instead of constructing a protocol from scratch and then proving its security using the Real-Ideal paradigm, we will use OT as a black-box and build our protocol on top of it. The protocol is as follows: Alice will play the role of a sender in OT protocol with the following two inputs: $\{m_0 = x_0 \wedge 0, m_1 = x_0 \wedge 1\}$, whereas Bob will play the role of a receiver in the OT protocol with choice bit $b = x_1$. Bob will thus receive m_b as the output of OT protocol, and then simply send over m_b to Alice. This protocol is illustrated in Figure 21.5.

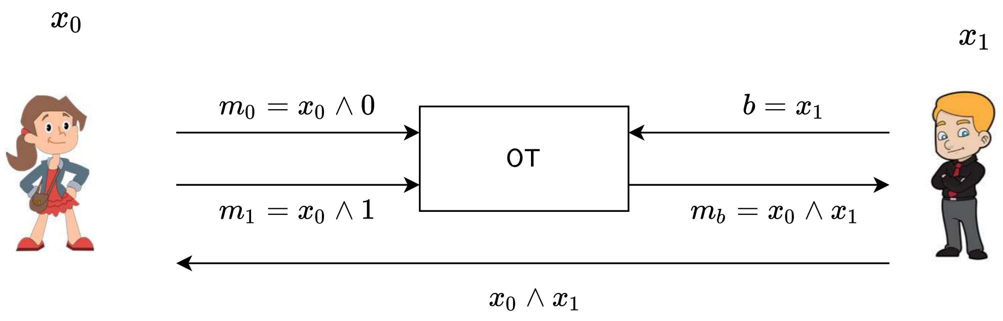


FIGURE 21.5: Protocol for securely computing the AND function

Now we will analyze this protocol: Suppose Bob's input $x_1 = 0$. Then, his choice bit $b = 0$ and therefore Bob will learn $m_0 = x_0 \wedge 0$ which matches the expected output of AND functionality. In case Bob' input $x_1 = 1$, then $b = 1$ and Bob will learn $m_1 = x_1 \wedge 1$ which again matches the expected output of AND functionality. Also, by the security of the underlying OT, in both cases Bob will learn only one of the messages, either m_0 or m_1 (corresponding to the correct output of AND functionality) and nothing else. Similarly, Alice will not learn anything about bob's input x_1 (which corresponds to the choice bit b in the underlying OT protocol) other than what is revealed by the output $x_0 \wedge x_1$.

- **Secure OR**: The protocol for secure OR is similar to the protocol we just saw for computing the AND functionality except for a small change: Instead of Alice providing $\{m_0 = x_0 \wedge 0, m_1 = x_0 \wedge 1\}$ as inputs to the underlying OT protocol, she will provide $\{m_0 = x_0 \vee 0, m_1 = x_0 \vee 1\}$ as inputs. The entire protocol diagram is shown in Figure 21.6. The correctness and security of this protocol follows essentially the same analysis as the protocol for Secure AND.

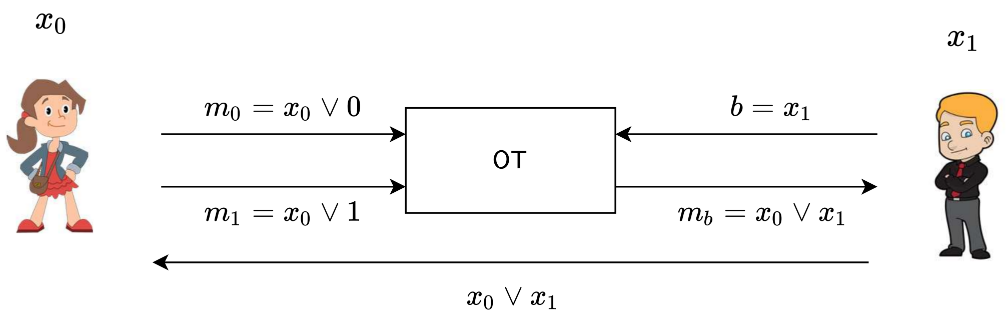


FIGURE 21.6: Protocol for securely computing the OR function

21.4 Secure Computation of General Circuits

Equipped with secure protocols for computing some basic functions, we can now proceed towards computing arbitrary boolean circuits composed of AND, OR, NOT gates. But before doing that, we will introduce a technique known as "Secret Sharing" which will help us towards our construction.

At a high-level, secret sharing is a technique for splitting a secret s into multiple pieces $s_1, s_2, \dots, s_n$ in such a way that individual pieces (or some subset of them) reveal no information about the secret s. However, when enough pieces are combined, the secret s can be efficiently recovered. A simple approach for performing such a splitting is the XOR secret sharing. In this scheme, a secret bit s is split into n pieces in the following way:

1. Sample $n - 1$ uniformly random bits and assign it to $s_1, \dots, s_{n-1}$
2. Set $s_n := s \oplus (s_1 \oplus s_2 \oplus \dots ..s_{n-1})$

This guarantees that any set of $n - 1$ pieces of s_i does not contain any information about the secret s. This is because any missing s_i piece(s) essentially acts as a One-Time-Pad and information theoretically hides s. However, if all n pieces $\{s_i\}_{i \in [n]}$ are known, then they can be XORed together to reconstruct the secret s. Fig 21.7 shows an illustration of the XOR based secret sharing scheme that we just described.

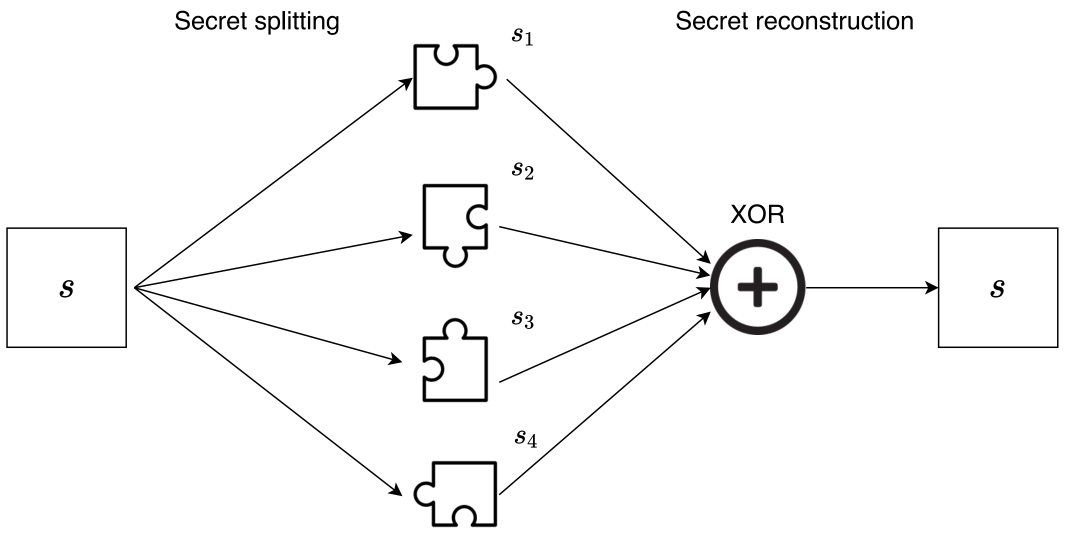


FIGURE 21.7: Secret sharing a secret s into $n = 4$ pieces using XOR secret sharing and then recombining the pieces to recover the original secret

There is fancier technique known as "Shamir secret sharing" [2] which allows a threshold-based splitting i.e. as long as less than a threshold t pieces (shares) are known, the pieces do not reveal any information about s. But any subset of t (or more) pieces are sufficient to completely reconstruct the original secret s. The value of t can be set to any value between 2 and n before performing the splitting (Note that the XOR based secret sharing we described only allows for $t = n$)

Now, equipped with this new technique, we are ready to devise a method for securely computing arbitrary boolean circuits (composed of AND, OR, NOT gates). The high-level idea is the following: For each gate G in the circuit with inputs a and b, we would like Alice and Bob to end up with an XOR secret sharing of $G(a, b)$ (instead of the actual output $G(a, b)$). The reason for this is *modularity* and *security*: The secret-shared outputs of a particular gate can then be used as inputs to the next gate in the circuit and the same process can be repeated till we reach the final output gate of the circuit. At that point, Alice and Bob can simply exchange their final secret share to retrieve the final output of the circuit. Also, such an approach ensures that the output of any intermediate gates in the overall circuit is completely hidden from the view of Alice and Bob. Indeed, this is essential for ensuring that no information about the inputs of parties is leaked during computation (beyond the final output of the circuit).

Having said that, we will now see how Alice and Bob, holding a secret sharing of a and b, can compute a secret sharing of the output $g = G(a, b)$ where $G \in \{\text{AND, OR, NOT}\}$. In more detail, suppose Alice holds $\{a_0, b_0\}$ and Bob holds $\{a_1, b_1\}$ s.t. $a_0 \oplus a_1 = a$ and $b_0 \oplus b_1 = b$. Then we want Alice to end up with g_0 and Bob to end up with g_1 so that $g_0 \oplus g_1 = g$. Fig. 21.8 illustrates the idea that we just described.

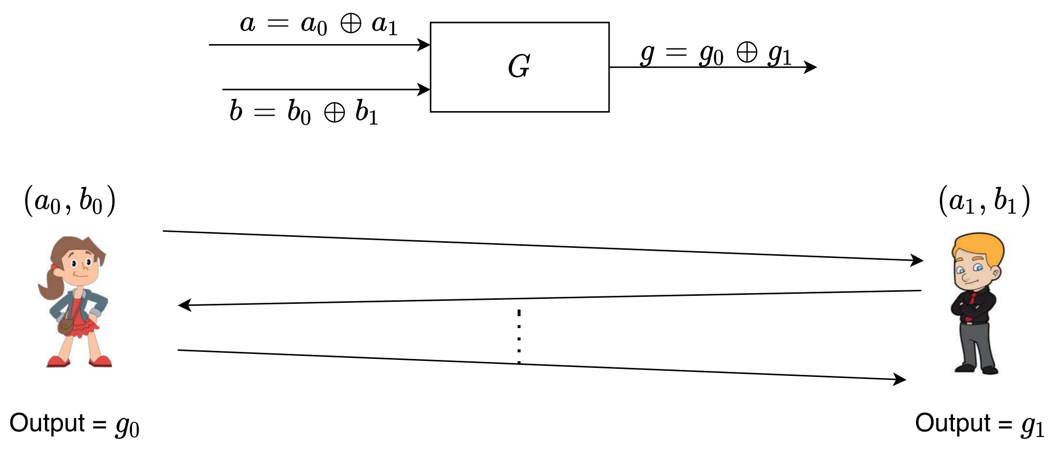


FIGURE 21.8: Secret sharing based computation of an arbitrary gate G

1. **XOR on secret shared data**: Suppose Alice and Bob hold a secret share of two values - a and b - and we want them to end up with a secret share of $a \oplus b$. In more detail, Alice holds (a_0, b_0) and Bob holds (a_1, b_1) such that $a_0 \oplus a_1 = a$ and $b_0 \oplus b_1 = b$, then we want Alice and Bob to end up with g_0 and g_1 respectively such that $g_0 \oplus g_1 = a \oplus b$. This can be easily done by just letting Alice and Bob locally XOR their secret shared values. In more detail, Alice computes $g_0 := a_0 \oplus b_0$ and Bob computes $g_1 := a_1 \oplus b_1$. It can be seen that computing g_0 and g_1 does indeed satisfy our requirements. Note that we did not need any cryptography in order to compute XOR on secret shared data.

2. **AND on secret shared data**: This is not as trivial as the previous XOR case. To see how we might go about computing AND on secret shared inputs so that Alice and Bob end up with a secret share of $a \wedge b$, let's start by expanding $a \wedge b$ in terms of secret shares $\{a_0, a_1, b_0, b_1\}$:

$$\begin{aligned} a \wedge b &= (a_0 \oplus a_1) \wedge (b_0 \oplus b_1) \\ &= (a_0 \wedge b_0) \oplus (a_0 \wedge b_1) \oplus (a_1 \wedge b_0) \oplus (a_1 \wedge b_1) \end{aligned}$$

   If we observe the R.H.S of above expression closely, we find that the first and fourth terms can be easily computed locally by Alice and Bob respectively. In more detail, Alice can locally perform $a_0 \wedge b_0$ to get the first term, and similarly, Bob can locally perform $a_1 \wedge b_1$ to get the fourth term. However, computing the second and third term is not that easy because the terms involve shares held by both parties jointly. In other words, neither Alice and Bob have enough data to compute either second term or the third term locally (unlike the case with first and fourth terms). Therefore, they must perform some kind of secure computation in order to compute the value of those terms. Specifically, we can use the OT based Secure AND protocol which we constructed earlier in order to compute the second and third terms. [4]

3. **NOT on secret shared data**: In this case, Alice and Bob hold a secret share of a single value a, and we want them to end up with a secret share of complement of a (denoted by $a' = 1 \oplus a$). In more detail, Alice holds a_0 and Bob holds a_1 such that $a_0 \oplus a_1 = a$, then we want Alice and Bob to end up with g_0 and g_1 respectively such that $g_0 \oplus g_1 = a'$. This can be easily done by just letting Alice locally invert/complement her secret share and let Bob retain her original share. In more detail, Alice computes $g_0 := 1 \oplus a_0$ and Bob computes $g_1 := a_1$. It can be seen that computing g_0 and g_1 in this fashion does indeed satisfy our requirements since $g_0 \oplus g_1 = 1 \oplus a_0 \oplus a_1 = 1 \oplus a$. Note that again, similar to XOR case, we did not need any cryptography in order to compute NOT of a secret shared data.

Having constructed our basic primitives for computing on secret-shared data, we can now describe how Alice and Bob would securely compute any arbitrary circuit composed of AND, OR, NOT gates. The steps are as follows:

1. Alice and Bob begin by XOR secret sharing each of their inputs to the circuit.

2. Alice and Bob compute each gate of the circuit using the secret shared inputs to obtain secret share of the output. The secret share of the output is then used as an input to the next gate in the circuit.

Fig 21.9 shows an illustration of the idea that we just described on a circuit composed of 3 gates. Alice and Bob begin by secret sharing their inputs to obtain secret shares to all the input wires to the circuit. Now they can perform the following steps:

- Alice using (a_0, b_0) and Bob using (a_1, b_1) can compute the gate G_1 in such a way that Alice ends up with c_0 and Bob ends up with c_1 where $c_0 \oplus c_1 = c$.

- Similarly, Alice using (A_0, B_0) and Bob using (A_1, B_1) can compute the gate G_2 in such a way that Alice ends up with C_0 and Bob ends up with C_1 where $C_0 \oplus C_1 = C$.

- Now Alice using (c_0, C_0) and Bob using (c_1, C_1) can compute the gate G_3 in such a way that Alice ends up with d_0 and Bob ends up with d_1 where $d_0 \oplus d_1 = D$.

Finally, Alice and Bob can send over d_0 and d_1 respectively and recover the final output as $d_0 \oplus d_1$.

Acknowledgement

These scribe notes were prepared by editing a light modification of the template designed by Alexander Sherstov.

[4]Alice and Bob would actually need to do some kind of re-randomization while computing the second and third terms using OT based AND protocol to make sure there is no leakage of partial information. We will revisit this in detail in the next lecture.

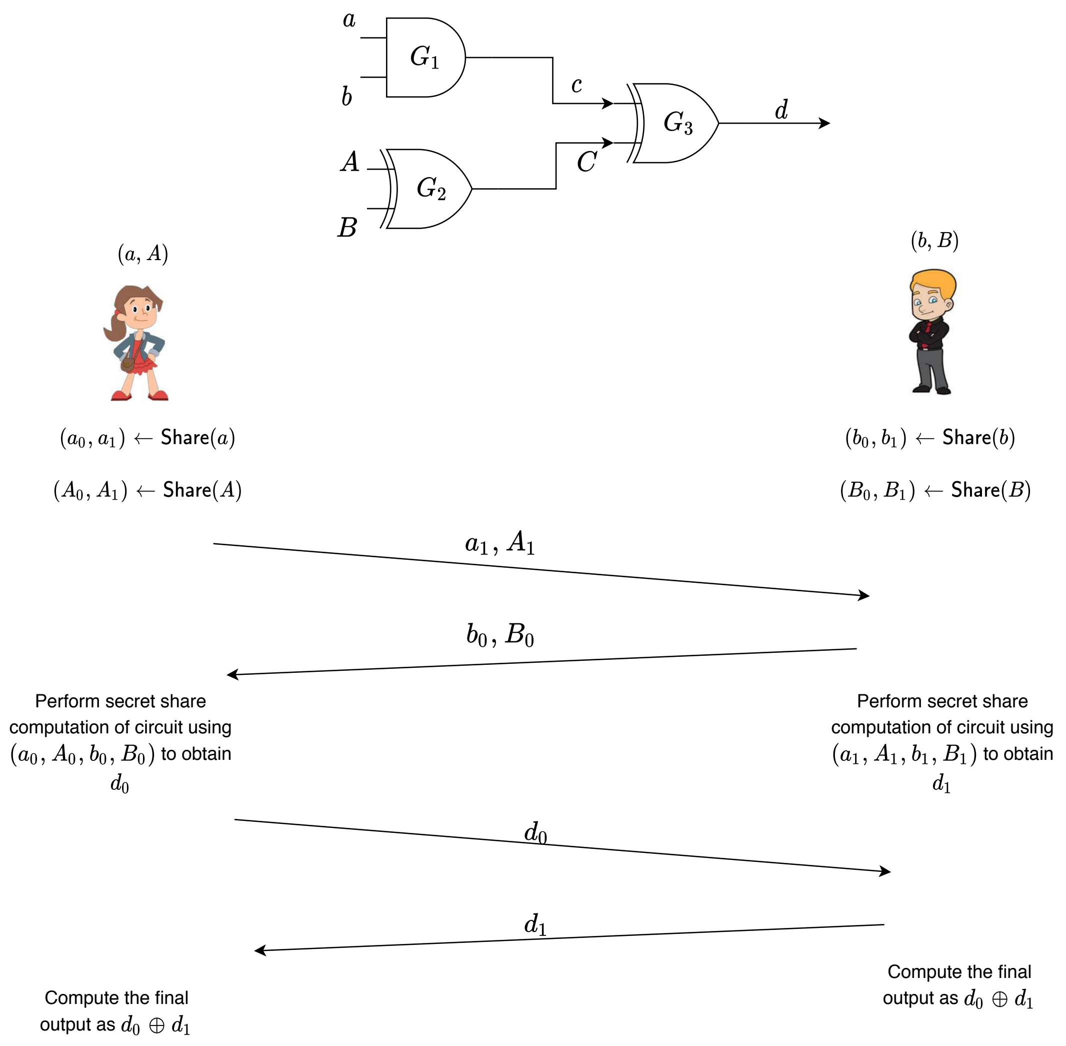

a
b
G_1
c
A
B
G_2
C
G_3
d
(a, A)
(b, B)
$(a_0, a_1) \leftarrow \mathsf{Share}(a)$
$(A_0, A_1) \leftarrow \mathsf{Share}(A)$
$(b_0, b_1) \leftarrow \mathsf{Share}(b)$
$(B_0, B_1) \leftarrow \mathsf{Share}(B)$
a_1, A_1
b_0, B_0
Perform secret share computation of circuit using (a_0, A_0, b_0, B_0) to obtain d_0
Perform secret share computation of circuit using (a_1, A_1, b_1, B_1) to obtain d_1
d_0
d_1
Compute the final output as $d_0 \oplus d_1$
Compute the final output as $d_0 \oplus d_1$

FIGURE 21.9: Secret sharing based computation of a sample circuit composed of 3 gates

References

[1] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[2] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.