CHAPTER 22

Secure Multi-party Computation

Chapter Goals

- To introduce the concept of multi-party computation.
- To present a two-party protocol based on Yao's garbled-circuit construction.
- To present a multi-party protocol based on Shamir secret sharing.

22.1. Introduction

Secure multi-party computation is an area of cryptography which deals with two or more parties computing a function on their private inputs. They wish to do so in a way that means that their private inputs still remain private. Of course depending on the function being computed, some information about the inputs may leak. The classical example is the so-called millionaires problem; suppose a bunch of millionaires have a lunch time meeting at an expensive restaurant and decide that the richest of them will pay the bill. However, they do not want to reveal their actual wealth to each other. This is an example of a secure multi-party computation. The inputs are the values x_i, which denote the wealth of each party, and the function to be computed is

$$f(x_1, \ldots, x_n) = i \text{ where } x_i > x_j \text{ for all } i \neq j.$$

Clearly, if we compute such a function, then some information about party i's value leaks; i.e. that it is greater than all the other values. However, we require in secure multi-party computation that this is the only information which leaks; even to the parties participating in the protocol.

One can consider a number of our previous protocols as being examples of secure multi-party computation. For example, the voting protocol given previously involves the computation of the result of each party voting, without anyone learning the vote being cast by a particular party. Encryption is a multi-party computation between three parties: the sender, the receiver and the adversary. Only the sender has an input (which is the message to be encrypted) and only the receiver has an output (which is the message when decrypted). In fact we can essentially see all of cryptography as some form of multi-party computation.

One solution to securely evaluating a function is for all the parties to send their inputs to a trusted third party. This trusted party then computes the function and passes the output back to the parties. However, we want to remove such a trusted third party entirely. Intuitively a multi-party computation is said to be secure if the information which is leaked is precisely that which would have leaked if the computation had been conducted by encrypting messages to a trusted third party.

This is not the only security issue that needs to be addressed when considering secure multi-party computation. There are two basic security models:

- In the first model the parties are guaranteed to follow the protocols, but are interested in breaking the privacy of their fellow participants. Such adversaries are called honest-but-curious, and they in some sense correspond to passive adversaries in other areas of

N.P. Smart, *Cryptography Made Simple*, Information Security and Cryptography, DOI 10.1007/978-3-319-21936-3_22

cryptography. Whilst honest-but-curious adversaries follow the protocol, a number of them could combine their different internal data so as to subvert the security of the non-corrupt parties.

- In the second model the adversaries can deviate from the protocol and may wish to pass incorrect data around so as to subvert the computation of the function. Again we allow such adversaries to talk to each other in a coalition. Such an adversary is called a malicious adversary. In such situations we would like the protocol to still complete, and compute the correct function, i.e. it should be both *correct* and *robust*. In this book we will not discuss modern protocols which trade robustness for other benefits.

There is a problem though. If we assume that communication is asynchronous, which is the most practically relevant situation, then some party must go last. In such a situation one party may have learnt the outcome of the computation, but one party may not have the value yet (namely the party which receives the last message). Any malicious party can clearly subvert the protocol by not sending the last message. Usually malicious adversaries are assumed not to perform such an attack. A protocol which is said to be secure against an adversary which can delete the final message is said to be fair.

In what follows we shall mainly explain the basic ideas behind secure multi-party computation in the case of honest-but-curious adversaries. We shall touch on the case of malicious adversaries for one of our examples though, as it provides a nice example of an application of various properties of Shamir secret sharing.

If we let n denote the number of parties which engage in the protocol, we would like to create protocols for secure multi-party computation which are able to tolerate a large number of corrupt parties. It turns out that there is a theoretical limit on the number of parties whose corruption can be tolerated.

- For the case of honest-but-curious adversaries we can tolerate fewer than $n/2$ corrupt parties, for computationally unbounded adversaries.
- If we restrict ourselves to computationally bounded adversaries then we can tolerate up to $n-1$ corrupt parties in the case of honest-but-curious adversaries.
- However, for a malicious adversary we can tolerate up to $n/3$ corrupt parties if we assume computationally unbounded adversaries.
- If we assume computationally bounded adversaries we can only tolerate less than $n/2$, unless we are prepared to accept an unfair/unrobust protocol in which case we can tolerate up to $n-1$ corrupt parties.

Protocols for secure multi-party computation usually fall into one of two distinct families. The first is based on an idea of Yao called a garbled circuit or Yao circuit: in this case one presents the function to be computed as a binary circuit, and then one “encrypts” the gates of this circuit to form the garbled circuit. This approach is clearly based on a computational assumption, i.e. that the encryption scheme is secure. The second approach is based on secret sharing schemes: here one usually represents the function to be computed as an arithmetic circuit. In this second approach one uses a perfectly secure secret sharing scheme to obtain perfect security.

It turns out that the first approach seems better suited to the case where there are two parties, whilst the second approach seems better suited to the case of three or more parties. In our discussion below we will present a computationally secure solution for the two-party case in the presence of honest-but-curious adversaries, based on Yao circuits. This approach can be extended to more than two parties and a malicious adversary, but doing this is beyond the scope of this book. We then present a protocol for the multi-party case which is perfectly secure. We sketch two versions, one which provides security against honest-but-curious adversaries and one which provides security against malicious adversaries.

22.2. The Two-Party Case

We shall in this section consider the method of secure multi-party computation based on garbled circuits. We suppose there are two parties A and B with inputs x and y respectively, and that A wishes to compute $f_A(x, y)$ and B wishes to compute $f_B(x, y)$. Recall this needs to be done without B learning anything about x or $f_A(x, y)$, except what he can deduce from $f_B(x, y)$ and y, with a similar privacy statement applying to A.

First note that it is enough for B to receive the output of a related function f. To see this we let A have an extra secret input k which is as long as the maximum output of her function $f_A(x, y)$. If we can create a protocol in which B learns the value of the function

$$f(x, y, k) = (k \oplus f_A(x, y), f_B(x, y)),$$

then B simply sends the value of $k \oplus f_A(x, y)$ back to A who can then decrypt it using k, and so determine $f_A(x, y)$. Hence, we will assume that there is only one function which needs to be computed and that its output will be determined by B.

So suppose $f(x, y)$ is the function which is to be computed; we will assume that $f(x, y)$ can be computed in polynomial time. Therefore there is also a polynomial-sized binary circuit which will also compute the output of the function. In the forthcoming example we will write out such a circuit, and so in Figure 22.1 we recall the standard symbols for a binary circuit.

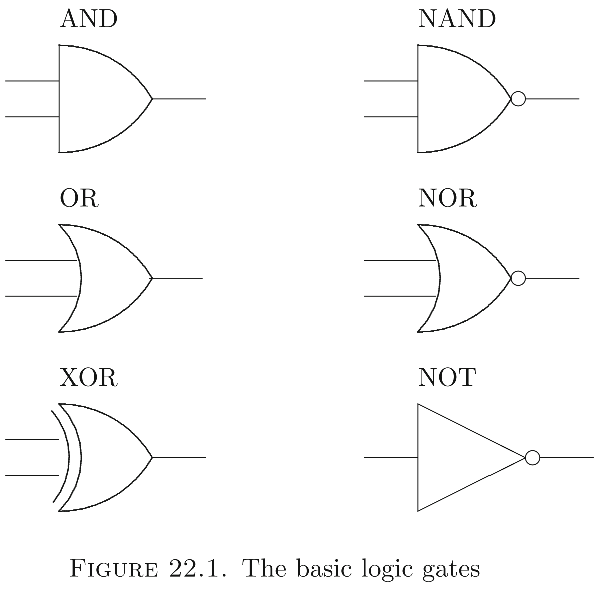

Figure 22.1. The basic logic gates

A binary circuit can be represented by a collection of wires $W = \{w_1, \ldots, w_n\}$ and a collection of gates $G = \{g_1, \ldots, g_m\}$. Each gate is a function which takes as input the values of two wires, and produces the value of the output wire. For example suppose g_1 is an AND gate which takes as input wires w_1 and w_2 and produces the output wire w_3. Then gate g_1 can be represented by the following truth table.

w_1	w_2	w_3
0	0	0
0	1	0
1	0	0
1	1	1

In other words, the gate g_i represents a function, $w_3 \leftarrow g_i(w_1, w_2)$ such that

$$0 = g_i(0,0) = g_i(1,0) = g_i(0,1) \text{ and } 1 = g_i(1,1).$$

22.2.1. Garbled Circuit Construction: In Yao's protocol one party constructs a garbled circuit (we shall call this party A), the other party evaluates the garbled circuit (we shall call this party B). The garbled circuit is constructed as follows:

- For each wire w_i two random cryptographic keys are selected, k_i^0 and k_i^1. The first one represents the encryption of the zero value and the second represents the encryption of the one value.
- For each wire a random value $\rho_i \in \{0,1\}$ is chosen. This is used to also encrypt the actual wire value. If the actual wire value is v_i then the encrypted, or "external" value, is given by $e_i = v_i \oplus \rho_i$.
- For each gate we compute a "garbled table" representing the function of the gate on these encrypted values. Suppose g_i is a gate with input wires w_{i_0} and w_{i_1} and output wire w_{i_2}, then the garbled table is the following four values, for some encryption function E:

  $$c_{a,b}^{w_{i_2}} = E_{k_{w_{i_0}}^{a\oplus\rho_{i_0}}, k_{w_{i_1}}^{b\oplus\rho_{i_1}}}\left(k_{w_{i_2}}^{o_{a,b}} \| (o_{a,b} \oplus \rho_{i_2})\right) \text{ for } a, b \in \{0,1\}.$$

  where $o_{a,b} = g_i(a \oplus \rho_{i_0}, b \oplus \rho_{i_1})$.

We do not consider exactly what encryption function is chosen; such a discussion is slightly beyond the scope of this book. If you want further details then look in the Further Reading section at the end of this chapter, or just assume we take an encryption scheme which is suitably secure.

The above may seem rather confusing so we illustrate the method for constructing the garbled circuit with an example. Suppose A and B each have as input two bits; we shall denote A's input wires by w_1 and w_2, whilst we shall denote B's input wires by w_3 and w_4. Suppose they now wish to engage in a secure multi-party computation so that B learns the value of the function

$$f(\{w_1, w_2\}, \{w_3, w_4\}) = (w_1 \wedge w_3) \vee (w_2 \oplus w_4).$$

A circuit to represent this function is given in Figure 22.2.

In Figure 22.2 we also present the garbled values of each wire and the corresponding garbled tables representing each gate. In this example we have the following values of ρ_i:

$$\rho_1 = \rho_4 = \rho_6 = \rho_7 = 1 \text{ and } \rho_2 = \rho_3 = \rho_5 = 0.$$

Consider the first wire; the two garbled values of the wire are k_1^0 and k_1^1, which represent the 0 and 1 values, respectively. Since $\rho_1 = 1$, the external value of the internal 0 value is 1 and the external value of the internal 1 value is 0. Thus we represent the garbled value of the wire by the pair of pairs

$$(k_1^0 \| 1, k_1^1 \| 0).$$

Now we look at the gates, and in particular consider the first gate. The first gate is an AND gate which takes as input the first and third wires. The first entry in this table corresponds to $a = b = 0$. Now the ρ values for the first and third wires are 1 and 0 respectively. Hence, the first entry in the table corresponds to what should happen if the keys k_1^1 and k_3^0 are seen, since

$$1 = 1 \oplus 0 = \rho_1 \oplus a \text{ and } 0 = 0 \oplus 0 = \rho_3 \oplus b.$$

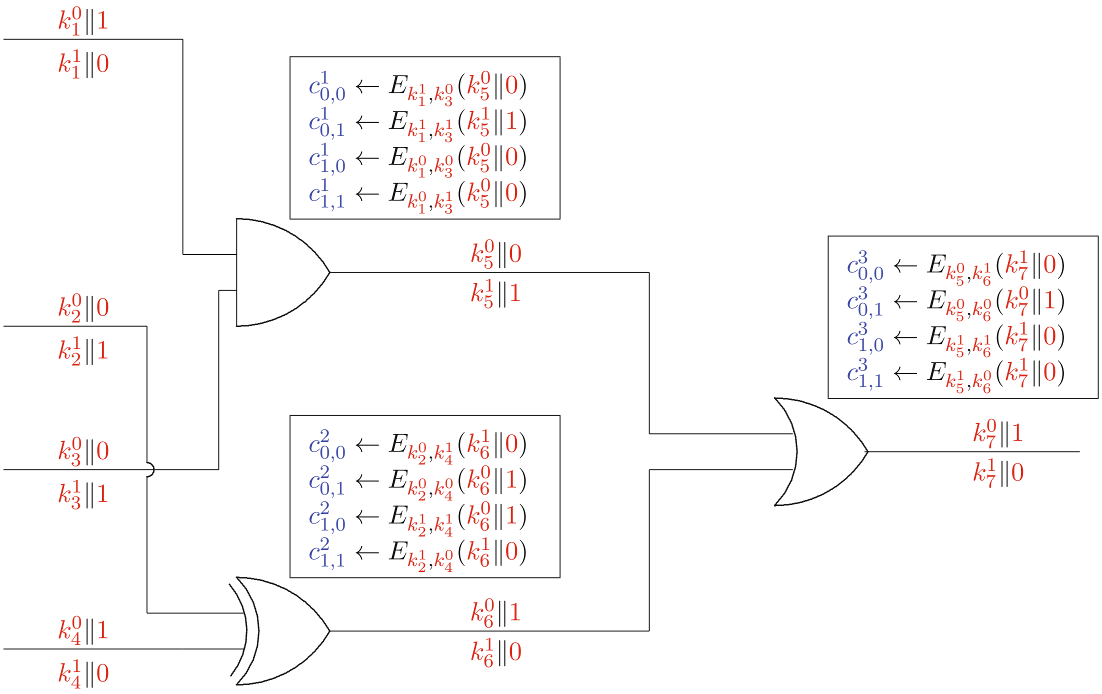
$k_1^0\|1$
$k_1^1\|0$
$c_{0,0}^1 \leftarrow E_{k_1^1,k_3^0}(k_5^0\|0)$
$c_{0,1}^1 \leftarrow E_{k_1^1,k_3^1}(k_5^1\|1)$
$c_{1,0}^1 \leftarrow E_{k_1^0,k_3^0}(k_5^0\|0)$
$c_{1,1}^1 \leftarrow E_{k_1^0,k_3^1}(k_5^0\|0)$
$k_5^0\|0$
$k_5^1\|1$
$c_{0,0}^3 \leftarrow E_{k_5^0,k_6^1}(k_7^1\|0)$
$c_{0,1}^3 \leftarrow E_{k_5^0,k_6^0}(k_7^0\|1)$
$c_{1,0}^3 \leftarrow E_{k_5^1,k_6^1}(k_7^1\|0)$
$c_{1,1}^3 \leftarrow E_{k_5^1,k_6^0}(k_7^1\|0)$
$k_2^0\|0$
$k_2^1\|1$
$k_3^0\|0$
$k_3^1\|1$
$c_{0,0}^2 \leftarrow E_{k_2^0,k_4^1}(k_6^1\|0)$
$c_{0,1}^2 \leftarrow E_{k_2^0,k_4^0}(k_6^0\|1)$
$c_{1,0}^2 \leftarrow E_{k_2^1,k_4^1}(k_6^0\|1)$
$c_{1,1}^2 \leftarrow E_{k_2^1,k_4^0}(k_6^1\|0)$
$k_7^0\|1$
$k_7^1\|0$
$k_6^0\|1$
$k_6^1\|0$
$k_4^0\|1$
$k_4^1\|0$

FIGURE 22.2. A garbled circuit

Now the AND gate should produce the 0 output on input of 1 and 0, thus the thing which is encrypted in the first line is the key representing the zero value of the fifth wire, i.e. k_5^0, plus the "external value" of 0, namely $0 = 0 \oplus 0 = 0 \oplus \rho_5$.

22.2.2. Garbled Circuit Evaluation: We now describe how the circuit is evaluated by party B. Please refer to Figure 22.3 for a graphical description of this. We assume that B has obtained in some way the specific garbled values of the input wires marked in blue in Figure 22.3, and the value of ρ_i for the output wires; in our example this is just ρ_7. Firstly party B evaluates the AND gate; he knows that the external value of wire one is 1 and the external value of wire three is 1. Thus he looks up the entry $c_{1,1}^1$ in the table and decrypts it using the two keys he knows, i.e. k_1^0 and k_3^1. He then obtains the value $k_5^0\|0$. He has no idea whether this represents the zero or one value of the fifth wire, since he has no idea as to the value of ρ_5.

Party B then performs the same operation with the exclusive-or gate. This has input wire 2 and wire 4, for which party B knows that the external values are 0 and 1 respectively. Thus party B decrypts the entry $c_{0,1}^2$ to obtain $k_6^0\|1$. A similar procedure is then carried out with the final OR gate, using the keys and external values of the fifth and sixth wires. This results in a decryption which reveals the value $k_7^0\|1$. So the external value of the seventh wire is equal to 1, but party B has been told that $\rho_7 = 1$, and hence the internal value of wire seven will be $0 = 1 \oplus 1$. Hence, the output of the function is the bit 0.

22.2.3. Yao's Protocol: We are now in a position to describe Yao's protocol in detail. The protocol proceeds in five phases as follows:

(1) Party A generates the garbled circuit as above, and transmits to party B only the values $c_{a,b}^i$.

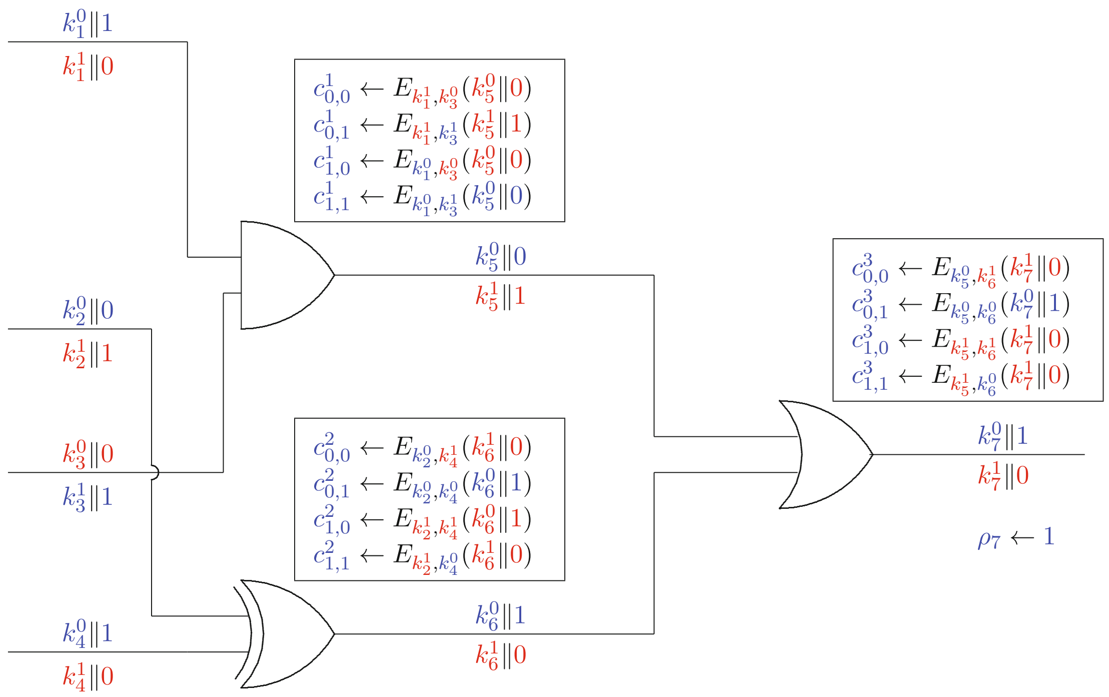
$k_1^0\|1$
$k_1^1\|0$
$k_2^0\|0$
$k_2^1\|1$
$k_3^0\|0$
$k_3^1\|1$
$k_4^0\|1$
$k_4^1\|0$
$c^1_{0,0} \leftarrow E_{k_1^1,k_3^0}(k_5^0\|0)$
$c^1_{0,1} \leftarrow E_{k_1^1,k_3^1}(k_5^1\|1)$
$c^1_{1,0} \leftarrow E_{k_1^0,k_3^0}(k_5^0\|0)$
$c^1_{1,1} \leftarrow E_{k_1^0,k_3^1}(k_5^0\|0)$
$k_5^0\|0$
$k_5^1\|1$
$c^2_{0,0} \leftarrow E_{k_2^0,k_4^1}(k_6^1\|0)$
$c^2_{0,1} \leftarrow E_{k_2^0,k_4^0}(k_6^0\|1)$
$c^2_{1,0} \leftarrow E_{k_2^1,k_4^1}(k_6^0\|1)$
$c^2_{1,1} \leftarrow E_{k_2^1,k_4^0}(k_6^1\|0)$
$k_6^0\|1$
$k_6^1\|0$
$c^3_{0,0} \leftarrow E_{k_5^0,k_6^1}(k_7^1\|0)$
$c^3_{0,1} \leftarrow E_{k_5^0,k_6^0}(k_7^0\|1)$
$c^3_{1,0} \leftarrow E_{k_5^1,k_6^1}(k_7^1\|0)$
$c^3_{1,1} \leftarrow E_{k_5^1,k_6^0}(k_7^1\|0)$
$k_7^0\|1$
$k_7^1\|0$
$\rho_7 \leftarrow 1$

FIGURE 22.3. Evaluating a garbled circuit

(2) Party A then transmits to party B the garbled values of the component of its input wires. For example, suppose that party A's input is $w_1 = 0$ and $w_2 = 0$. Then party A transmits to party B the two values $k_1^0\|1$ and $k_2^0\|0$. Note that party B cannot learn the actual values of w_1 and w_2 from these values since he does not know ρ_1 and ρ_2, and the keys k_1^0 and k_2^0 just look like random keys.
(3) Party A and B then engage in an oblivious transfer protocol, as in Section 20.3, for each of party B's input wires. In our example suppose that party B's input is $w_3 = 1$ and $w_4 = 0$. The two parties execute two oblivious transfer protocols, one with A's input $k_3^0\|0$ and $k_3^1\|1$, and B's input 1, and one with A's input $k_4^0\|1$ and $k_4^1\|0$, and B's input 0. At the end of this oblivious transfer phase party B has learnt $k_3^1\|1$ and $k_4^0\|1$.
(4) Party A then transmits to party B the values of ρ_i for all of the output wires. In our example he reveals the value of $\rho_7 = 1$.
(5) Finally party B evaluates the circuit using the garbled input wire values he has been given, using the technique described above.

In summary, in the first stage all party B knows about the garbled is in the blue items in Figure 22.2, but by the last stage he knows the blue items in Figure 22.3.

In our example we can now assess what party B has learnt from the computation. Party B knows that the output of the final OR gate is zero, which means that the inputs must also be zero, which means that the output of the AND gate is zero and the output of the exclusive-or gate is zero. However, party B already knew that the output of the AND gate will be zero, since his own input was zero. However, party B has learnt that party A's second input wire represented zero, since otherwise the exclusive-or gate would not have output zero. So whilst party A's first input remains private, the second input does not. This is what we meant by a protocol keeping the inputs private, bar what could be deduced from the output of the function.

22.3. The Multi-party Case: Honest-but-Curious Adversaries

The multi-party case is based on using a secret sharing scheme to evaluate an arithmetic circuit. An arithmetic circuit consists of a finite field $\mathbb{F}_q$ and a polynomial function (which could have many inputs and outputs) defined over the finite field. The idea is that such a function can be evaluated by executing a number of addition and multiplication gates over the finite field.

Given an arithmetic circuit it is clear one could express it as a binary circuit, by simply expanding out the addition and multiplication gates of the arithmetic circuit as their binary circuit equivalents. One can also represent every binary circuit as an arithmetic circuit, since every gate in the binary circuit can be represented as a linear function of the input values to the gate and their products. For example, suppose we represent the binary values 0 and 1 by 0 and 1 in the finite field $\mathbb{F}_q$, and that the characteristic of $\mathbb{F}_q$ is larger than two. We then have that the binary exclusive-or gate can be written as $x \oplus y = -2 \cdot x \cdot y + x + y$ over $\mathbb{F}_q$, and the binary "and" gate can be written as $x \odot y = x \cdot y$.

Whilst the two representations are equivalent it is clear that some functions are easier to represent as binary circuits and some are easier to represent as arithmetic circuits.

As before we shall present the protocol via a running example. We shall suppose we have six parties $P_1, \ldots, P_6$ who have six secret values $x_1, \ldots, x_6$, each of which lie in $\mathbb{F}_p$, for some reasonably large prime p. For example we could take $p \approx 2^{128}$, but in our example to make things easier to represent we will take $p = 101$. The parties are assumed to want to compute the value of the function

$$f(x_1, \ldots, x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \pmod{p}.$$

Hence, the arithmetic circuit for this function consists of three multiplication gates and two addition gates, as in Figure 22.4, where we label the intermediate values as numbered "wires".

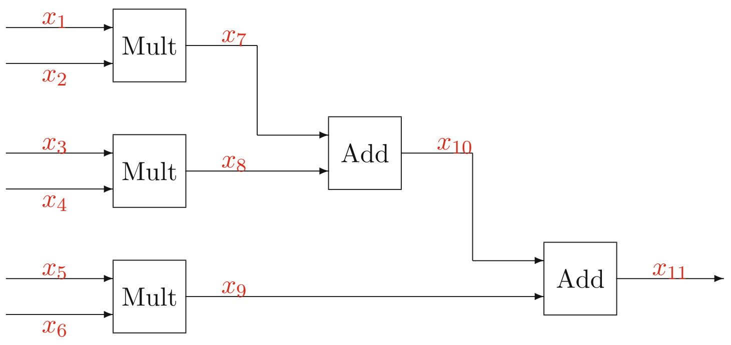

FIGURE 22.4. Graphical representation of the example arithmetic circuit

We will use Shamir's secret sharing, in which case our basic protocol is as follows: The value of each wire x_i is shared between all players, with each player j obtaining a share $x_i^{(j)}$. Clearly, if enough players come together, then they can determine the value of the wire x_i, by the properties of the secret sharing scheme.

Each player can create shares of his or her own input values at the start of the protocol and send a share to each player. Thus we need to show how to obtain the shares of the outputs of a gate, given shares of the inputs of the gate. Recall that in Shamir's secret sharing the shared value is given by the constant term of a polynomial f of degree t, with the shares being the evaluation of the polynomial at given positions corresponding to each participant $f(i)$.

22.3.1. Addition Gates: First we consider how to compute the `Add` gates. Suppose we have two secrets a and b which are shared using the polynomials

$$f(X) = a + f_1 \cdot X + \cdots + f_t \cdot X^t,$$
$$g(X) = b + g_1 \cdot X + \cdots + g_t \cdot X^t.$$

Each of our parties has a share $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$. Now consider the polynomial

$$h(X) = f(X) + g(X).$$

This polynomial provides a sharing of the sum $c = a + b$, and we have

$$c^{(i)} = h(i) = f(i) + g(i) = a^{(i)} + b^{(i)}.$$

Hence, the parties can compute a sharing of the output of an `Add` gate without any form of communication between them.

22.3.2. Multiplication Gates: Computing the output of a `Mult` gate is more complicated. First we recap the following property of Lagrange interpolation. If $f(X)$ is a polynomial and we distribute the values $f(i)$ then there is a vector $(r_1, \ldots, r_n)$, called the recombination vector, such that

$$f(0) = \sum_{i=1}^{n} r_i \cdot f(i).$$

And the same vector works for all polynomials $f(X)$ of degree at most $n - 1$.

To compute the `Mult` gate we perform the following four steps. We assume as input that each party has a share of a and b via $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$, where $f(0) = a$ and $g(0) = b$. We wish to compute a sharing $c^{(i)} = h(i)$ such that $h(0) = c = a \cdot b$.

- Each party locally computes $d^{(i)} = a^{(i)} \cdot b^{(i)}$.
- Each party produces a polynomial $\delta_i(X)$ of degree at most t such that $\delta_i(0) = d^{(i)}$.
- Each party i distributes to party j the value $d_{i,j} = \delta_i(j)$.
- Each party j computes $c^{(j)} = \sum_{i=1}^{n} r_i \cdot d_{i,j}$.

So why does this work? Consider the first step; here we are actually effectively computing a polynomial $h'(X)$ of degree at most $2 \cdot t$, with $d^{(i)} = h'(i)$, and $c = h'(0)$. Hence, the only problem with the sharing in the first step is that the underlying polynomial has too high a degree. The main thing to note is that if

$$2 \cdot t \leq n - 1 \tag{23}$$

then we have $c = \sum_{i=1}^{n} r_i \cdot d^{(i)}$. Now consider the polynomials $\delta_i(X)$ generated in the second step, and consider what happens when we recombine them using the recombination vector, i.e. set

$$h(X) = \sum_{i=1}^{n} r_i \cdot \delta_i(X).$$

Since the $\delta_i(X)$ are all of degree at most t, the polynomial $h(X)$ is also of degree at most t. We also have that

$$h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) = \sum_{i=1}^{n} r_i \cdot d^{(i)} = c,$$

assuming $2 \cdot t \leq n - 1$. Thus $h(X)$ is a polynomial which *could* be used to share the value of the product. Not only that, but it *is* the polynomial underlying the sharing produced in the final step. To see this notice that

$$h(j) = \sum_{i=1}^{n} r_i \cdot \delta_i(j) = \sum_{i=1}^{n} r_i \cdot d_{i,j} = c^{(j)}.$$

Example: So assuming $t < n/2$ we can produce a protocol which evaluates the arithmetic circuit correctly. We illustrate the method by examining what would happen for our example circuit in Figure 22.4, with $p = 101$. Recall that there are six parties; we shall assume that their inputs are given by

$$x_1 = 20,\ x_2 = 40,\ x_3 = 21,\ x_4 = 31,\ x_5 = 1,\ x_6 = 71.$$

Each party first computes a sharing ${x_i}^{(j)}$ of their secret amongst the six parties. They do this by each choosing a random polynomial of degree $t = 2$ and evaluating it at $j = 1, 2, 3, 4, 5, 6$. The values obtained are then distributed securely to each party. Hence, each party obtains its row of the following table.

	i					
j	1	2	3	4	5	6
1	44	2	96	23	86	83
2	26	0	63	13	52	79
3	4	22	75	62	84	40
4	93	48	98	41	79	10
5	28	35	22	90	37	65
6	64	58	53	49	46	44

As an exercise you should work out the associated polynomials corresponding to each column.

The parties then engage in the multiplication protocol so as to compute sharings of $x_7 = x_1 \cdot x_2$. They first compute their local multiplication, by each multiplying the first two elements in their row of the above table, then they form a sharing of this local multiplication. These sharings of six numbers between six parties are then distributed securely. In our example run each party, for this multiplication, obtains the sharings given by its column of the following table.

	j					
i	1	2	3	4	5	6
1	92	54	20	91	65	43
2	10	46	7	95	7	46
3	64	100	96	52	69	46
4	23	38	41	32	11	79
5	47	97	77	88	29	1
6	95	34	11	26	79	69

Each party then takes the six values obtained and recovers their share of the value of x_7. We find that the six shares of x_7 are given by

$${x_7}^{(1)} = 9,\ {x_7}^{(2)} = 97,\ {x_7}^{(3)} = 54,\ {x_7}^{(4)} = 82,\ {x_7}^{(5)} = 80,\ {x_7}^{(6)} = 48.$$

Repeating the multiplication protocol twice more we also obtain a sharing of x_8 as

$${x_8}^{(1)} = 26,\ {x_8}^{(2)} = 91,\ {x_8}^{(3)} = 38,\ {x_8}^{(4)} = 69,\ {x_8}^{(5)} = 83,\ {x_8}^{(6)} = 80,$$

and x_9 as

$${x_9}^{(1)} = 57,\ {x_9}^{(2)} = 77,\ {x_9}^{(3)} = 30,\ {x_9}^{(4)} = 17,\ {x_9}^{(5)} = 38,\ {x_9}^{(6)} = 93.$$

We are then left with the two addition gates to produce the sharings of the wires x_{10} and x_{11}. These are obtained by locally adding together the various shared values so that

$$x_{11}{}^{(1)} = x_7{}^{(1)} + x_8{}^{(1)} + x_9{}^{(1)} \pmod{101} = 9 + 26 + 57 \pmod{101} = 92,$$

etc. to obtain

$$x_{11}{}^{(1)} = 92,\ x_{11}{}^{(2)} = 63,\ x_{11}{}^{(3)} = 21,\ x_{11}{}^{(4)} = 67,\ x_{11}{}^{(5)} = 100,\ x_{11}{}^{(6)} = 19.$$

The parties then make public these shares, and recover the hidden polynomial, of degree $t = 2$, which produces these sharings, namely $7 + 41 \cdot X + 44 \cdot X^2$. Hence, the result of the multi-party computation is the value 7.

Now assume that more than t parties are corrupt, in the sense that they collude to try to break the privacy of the non-corrupted parties. The corrupt parties can now come together and recover any of the underlying secrets in the scheme, since we have used Shamir secret sharing using polynomials of degree at most t. It can be shown, using the perfect secrecy of the Shamir secret sharing scheme, that as long as no more than t parties are corrupt then the above protocol is perfectly secure.

However, it is only perfectly secure assuming all parties follow the protocol, i.e. we are in the honest-but-curious model. As soon as we allow parties to deviate from the protocol, they can force the honest parties to produce invalid results. To see this just notice that a dishonest party could simply produce an invalid sharing of its product in the second part of the multiplication protocol above.

22.4. The Multi-party Case: Malicious Adversaries

To produce a scheme which is secure against active adversaries either we need to force all parties to follow the protocol or we should be able to recover from errors which malicious parties introduce into the protocol. It is the second of these two approaches which we shall follow in this section, by using the error correction properties of the Shamir secret sharing scheme. As already remarked, the above protocol is not secure against malicious adversaries, due to the ability of an attacker to make the multiplication protocol output an invalid answer. To make the above protocol secure against malicious adversaries we make use of various properties of the Shamir secret sharing scheme.

The protocol runs in two stages: The preprocessing stage does not involve any of the secret inputs of the parties, it depends purely on the number of multiplication gates in the circuit. In the main phase of the protocol the circuit is evaluated as in the previous section, but using a slightly different multiplication protocol. Malicious parties can force the preprocessing stage to fail, however if it completes then the honest parties will be able to evaluate the circuit as required.

The preprocessing phase runs as follows. First, using the techniques from Chapter 19, a pseudo-random secret sharing scheme, PRSS, and a pseudo-random zero sharing scheme, PRZS, are set up. Then for each multiplication gate in the circuit we compute a random triple of sharings $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. This is done as follows:

- Using PRSS generate two random sharings, $a^{(i)}$ and $b^{(i)}$, of degree t.
- Using PRSS generate another random sharing $r^{(i)}$ of degree t.
- Using PRZS generate a sharing $z^{(i)}$, of degree $2 \cdot t$ of zero.
- Each party then locally computes $s^{(i)} = a^{(i)} \cdot b^{(i)} - r^{(i)} + z^{(i)}$. Note that this local computation will produce a degree $2 \cdot t$ sharing of the value $s = a \cdot b - r$.
- Then the players broadcast their values $s^{(i)}$ and try to recover s. Here we make use of the error detection properties of Reed–Solomon codes. If the number of malicious parties is bounded by $t < n/3$, then any error in the degree-$(2 \cdot t)$ sharing will be detected. At this stage the parties abort the protocol if any error is found.
- Now the players locally compute the shares $c^{(i)}$ from $c^{(i)} = s + r^{(i)}$.

Assuming the above preprocessing phase completes successfully, all we need to do is specify how the parties implement a `Mult` in the presence of malicious adversaries. We assume the inputs to the multiplication gate are given by $x^{(i)}$ and $y^{(i)}$ and we wish to compute a sharing $z^{(i)}$ of the product $z = x \cdot y$. From the preprocessing stage, the parties also have for each gate, a triple of shares $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. The protocol for the multiplication is then as follows:

- Compute locally, and then broadcast, the values $d^{(i)} = x^{(i)} - a^{(i)}$ and $e^{(i)} = y^{(i)} - b^{(i)}$.
- Reconstruct the values of $d = x - a$ and $e = y - b$.
- Locally compute the shares $z^{(i)} = d \cdot e + d \cdot b^{(i)} + e \cdot a^{(i)} + c^{(i)}$.

Note that the reconstruction in the second step can be completed, even if the corrupt parties transmit invalid values, as long as there are at most $t < n/3$ malicious parties. Due to the error correction properties of Reed–Solomon codes, we can recover from any errors introduced by malicious parties. The above protocol produces a valid sharing of the output of the multiplication gate because

$$\begin{aligned} d \cdot e + d \cdot b + e \cdot a + c &= (x - a) \cdot (y - b) + (x - a) \cdot b + (y - b) \cdot a + c \\ &= ((x - a) + a) \cdot ((y - b) + b) \\ &= x \cdot y = z. \end{aligned}$$

Chapter Summary

- We have explained how to perform two-party secure computation, in the case of honest-but-curious adversaries, using Yao's garbled-circuit construction.
- For the multi-party case we have presented a protocol based on evaluating arithmetic, as opposed to binary, circuits which is based on Shamir secret sharing.
- The main issue with this latter protocol is how to evaluate the multiplication gates. We presented two methods: The first, simpler, method is applicable when one is only dealing with honest-but-curious adversaries, the second, more involved, method is for the case of malicious adversaries.

Further Reading

The original presentation of Yao's idea was apparently given in the talk which accompanied the paper in FOCS 1986, however the paper contains no explicit details of the protocol. It can be transformed into a scheme for malicious adversaries using a general technique of Goldreich et al. The discussion of the secret sharing based solution for the honest and malicious cases closely follows the treatment in Damgård et al.

I. Damgård, M. Geisler, M. Krøigaard and J.B. Nielsen. *Asynchronous multiparty computation: Theory and implementation* In Public Key Cryptography – PKC 2009, LNCS 5443, 160–179, Springer, 2009.

O. Goldreich, S. Micali and A. Wigderson. *How to play any mental game.* In Symposium on Theory of Computing – STOC 1987, 218–229, ACM, 1987.

A.C. Yao. *How to generate and exchange secrets.* In Foundations of Computer Science – FOCS 1986, 162–167, IEEE, 1986.