3

Fundamental MPC Protocols

In this chapter we survey several important MPC approaches, covering the main protocols and presenting the intuition behind each approach.

All of the approaches discussed can be viewed as a form of computing under encryption, or, more specifically, as secret-sharing the input data and computing on the shares. For example, an encryption $\mathsf{Enc}_k(m)$ of a message m with a key k can be seen as secret-sharing m, where one share is k and the other is $\mathsf{Enc}_k(m)$. We present several fundamental protocols illustrating a variety of generic approaches to secure computation, as summarized in Table 3.1. All of the protocols of this section target the semi-honest adversary model (Section 2.3.2). We discuss malicious-secure variants in Chapter 6. All of these

protocol	# parties	# rounds	circuit
Yao's GC (Section 3.1)	2	constant	Boolean
GMW (Section 3.2)	many	circuit depth	Boolean or arithmetic
BGW (Section 3.3)	many	circuit depth	Boolean or arithmetic
BMR (Section 3.5)	many	constant	Boolean
GESS (Section 3.6)	2	constant	Boolean *formula*

Table 3.1: Summary of semi-honest MPC protocols discussed in this chapter.

protocols build on oblivious transfer, which we discuss how to implement efficiently in Section 3.7.

3.1 Yao's Garbled Circuits Protocol

Yao's Garbled Circuits protocol (GC) is the most widely known and celebrated MPC technique. It is usually seen as best-performing, and many of the protocols we cover build on Yao's GC. While not having the best known communication complexity, it runs in constant rounds and avoids the costly latency associated with approaches, such as GMW (described in Section 3.2), where the number of communication rounds scales with the circuit depth.

3.1.1 GC Intuition

The main idea behind Yao's GC approach is quite natural. Recall, we wish to evaluate a given function $\mathcal{F}(x, y)$ where party P_1 holds $x \in X$ and P_2 holds $y \in Y$. Here X and Y are the respective domains for the inputs of P_1 and P_2.

Function as a look-up table. First, let's consider a function $\mathcal{F}$ for which the input domain is small and we can efficiently enumerate all possible input pairs, (x, y). The function $\mathcal{F}$ can be represented as a look-up table T, consisting of $|X| \cdot |Y|$ rows, $T_{x,y} = \langle \mathcal{F}(x, y) \rangle$. The output of $\mathcal{F}(x, y)$ is obtained simply by retrieving $T_{x,y}$ from the corresponding row.

This gives us an alternative (and much simplified!) view of the task at hand. Evaluating a look-up table can be done as follows. P_1 will encrypt T by assigning a randomly-chosen strong key to *each* possible input x and y. That is, for each $x \in X$ and each $y \in Y$, P_1 will choose $k_x \in_R \{0,1\}^\kappa$ and $k_y \in_R \{0,1\}^\kappa$. It will then encrypt T by encrypting each element $T_{x,y}$ of T with *both* keys k_x and k_y, and send the encrypted (and randomly permuted!) table $\langle \mathsf{Enc}_{k_x,k_y}(T_{x,y}) \rangle$ to P_2.

Now our task is to enable P_2 to decrypt (only) the entry $T_{x,y}$ corresponding to players' inputs. This is done by having P_1 send to P_2 the keys k_x and k_y. P_1 knows its input x, and hence simply sends key k_x to P_2. The key k_y is sent to P_2 using a 1-out-of-$|Y|$ Oblivious Transfer (Section 2.4). Once P_2 receives k_x and k_y, it can obtain the output $\mathcal{F}(x, y)$ by decrypting $T_{x,y}$ using those keys. Importantly, no other information is obtained by P_2. This is because P_2 only

has a single pair of keys, which can only be used to open (decrypt) a single table entry. We stress that, in particular, it is important that neither partial key, k_x or k_y, by itself can be used to obtain partial decryptions or even determine whether the partial key was used in the obtaining a specific encryption.[1]

Point-and-Permute. A careful reader may wonder how P_2 knows which row of the table T to decrypt, as this information is dependent on the inputs of both parties, and, as such, is sensitive.

The simplest way to address this is to encode some additional information in the encrypted elements of T. For example, P_1 may append a string of σ zeros to each row of T. Decrypting the wrong row with high probability ($p = \frac{1}{2^\sigma}$) will produce an entry which will not end with σ zeros, and hence will be rejected by P_2.

While the above approach works, it is inefficient for P_2, who expects to need to decrypt half of the rows of the table T. A much better approach, often called *point-and-permute*,[2] was introduced by Beaver *et al.* (1990). The idea is to interpret part of the key (namely, the last $\lceil \log |X| \rceil$ bits of the first key and the last $\lceil \log |Y| \rceil$ bits of the second key) as a pointer to the permuted table T, where the encryption will be placed. To avoid collisions in table row allocation, P_1 must ensure that the pointer bits don't collide within the space of keys k_x or within the space of k_y; this can be done in a number of ways. Finally, strictly speaking, key size must be maintained to achieve the corresponding level of security. As a consequence, rather than viewing key bits as a pointer, players will append the pointer bits to the key and maintain the desired key length.

In the subsequent discussions, we assume that the evaluator knows which row to decrypt. In protocol presentations, we may or may not explicitly include the point-and-permute component, depending on context.

[1] Consider a counter-example. Suppose P_2 was able to determine that a key k_x that it received from P_1 was used in encrypting r_x rows. Because some input combinations may be invalid, the encrypted look-up table T may have a unique number of rows relying on k_x, which will reveal x to P_2, violating the required security guarantees.

[2] This technique was not given a name by Beaver *et al.* (1990). Rather, this name came to be widely used by the community around 2010, as GC research progressed and need for a name arose. This technique is different from the *permute and point* technique introduced and coined in the information-theoretic garbled circuit construction of Kolesnikov (2005), which we discuss in Section 3.6.

Managing look-up table size. Clearly, the above solution is inefficient as it scales linearly with the domain size of $\mathcal{F}$. At the same time, for small functions, such as those defined by a single Boolean circuit gate, the domain has size 4, so using a look-up table is practical.

The next idea is to represent $\mathcal{F}$ as a Boolean circuit C and evaluate each gate using look-up tables of size 4. As before, P_1 generates keys and encrypts look-up tables, and P_2 applies decryption keys without knowing what each key corresponds to. However, in this setting, we cannot reveal the plaintext output of intermediate gates. This can be hidden by making the gate output also a key whose corresponding value is unknown to the evaluator, P_2.

For each wire w_i of C, P_1 assigns two keys k_i^0 and k_i^1, corresponding to the two possible values on the wire. We will refer to these keys as *wire labels*, and to the plaintext wire values simply as *wire values*. During the execution, depending on the inputs to the computation, each wire will be associated with a specific plaintext value and a corresponding wire label, which we will call *active value* and *active label*. We stress that the evaluator can know only the active *label*, but not its corresponding *value*, and not the *inactive* label.

Then, going through C, for each gate G with input wires w_i and w_j, and output wire w_t, P_1 builds the following encrypted look-up table:

$$T_G = \begin{pmatrix} \mathsf{Enc}_{k_i^0,k_j^0}(k_t^{G(0,0)}) \\ \mathsf{Enc}_{k_i^0,k_j^1}(k_t^{G(0,1)}) \\ \mathsf{Enc}_{k_i^1,k_j^0}(k_t^{G(1,0)}) \\ \mathsf{Enc}_{k_i^1,k_j^1}(k_t^{G(1,1)}) \end{pmatrix}$$

For example, if G is an AND gate, the look-up table will be:

$$T_G = \begin{pmatrix} \mathsf{Enc}_{k_i^0,k_j^0}(k_t^0) \\ \mathsf{Enc}_{k_i^0,k_j^1}(k_t^0) \\ \mathsf{Enc}_{k_i^1,k_j^0}(k_t^0) \\ \mathsf{Enc}_{k_i^1,k_j^1}(k_t^1) \end{pmatrix}$$

Each cell of the look-up table encrypts the *label corresponding to the output computed by the gate*. Crucially, this allows the evaluator P_2 to obtain the intermediate active labels on internal circuit wires and use them in the evaluation of $\mathcal{F}$ under encryption without ever learning their semantic value.

P_1 permutes the entries in each of the look-up tables (usually called *garbled tables* or *garbled gates*), and sends all the tables to P_2. Additionally, P_1 sends (only) the active labels of all wires corresponding to the input values to P_2. For input wires belonging to P_1's inputs to $\mathcal{F}$, this is done simply by sending the wire label keys. For wires belonging to P_2's inputs, this is done via 1-out-of-2 Oblivious Transfer.

Upon receiving the input keys and garbled tables, P_2 proceeds with the evaluation. As discussed above, P_2 must be able to decrypt the correct row of each garbled gate. This is achieved by the point-and-permute technique described above. In our case of a 4-row garbled table, the point-and-permute technique is particularly simple and efficient — one pointer bit is needed for each input, so there are two total pointer bits added to each entry in the garbled table. Ultimately, P_2 completes evaluation of the garbled circuit and obtains the keys corresponding to the output wires of the circuit. These could be sent to P_1 for decryption, thus completing the private evaluation of $\mathcal{F}$.

We note that a round of communication may be saved and sending the output labels by P_2 for decryption by P_1 can be avoided. This can be done simply by P_1 including the decoding tables for the output wires with the garbled circuit it sends. The decoding table is simply a table mapping each label on each *output* wire to its semantics (i.e. the corresponding plaintext value. Now, P_2 obtaining the output labels will look them up in the decoding table and obtain the output in plaintext.

At an intuitive level, at least, it is easy to see that this circuit-based construction is secure in the semi-honest model. Security against a corrupt P_1 is easy, since (other than the OT, which we assume has been separately shown to satisfy the OT security definition) that party receives no messages in the protocol! For a corrupt P_2, security boils down to the observation that the evaluator P_2 never sees both labels for the same wire. This is obviously true for the input wires, and it holds inductively for all intermediate wires (knowing only one label on each incoming wire of the gate, the evaluator can only decrypt one ciphertext of the garbled gate). Since P_2 does not know the correspondence between plaintext values and the wire labels, it has no information about the plaintext values on the wires, except for the output wires where the association between labels and values is explicitly provided by P_1. To simulate P_2's view, the simulator $\mathsf{Sim}_{\mathsf{P}_2}$ chooses random active labels for

each wire, simulates the three "inactive" ciphertexts of each garbled gate as dummy ciphertexts, and produces decoding information that decodes the active output wires to the function's output.

3.1.2 Yao's GC Protocol

Figure 3.1 formalizes Yao's gate generation, and Figure 3.2 summarizes Yao's GC protocol. For simplicity of presentation, we describe the protocol variant based on Random Oracle (defined in Section 2.2), even though a weaker assumption (the existence of pseudo-random functions) is sufficient for Yao's GC construction. The Random Oracle, denoted by H, is used in implementing garbled row encryption. We discuss different methods of instantiating H in Section 4.1.4. The protocol also uses Oblivious Transfer, which requires public-key cryptography.

For each wire label, a pointer bit, p_i, is added to the wire label key following the point-and-permute technique described in Section 3.1.1. The pointer bits leak no information since they are selected randomly, but they allow the evaluator to determine which row in the garbled table to decrypt, based on the pointer bits for the two active wires it has for the inputs. In Section 4.1 we discuss several ways for making Yao's GC protocol more efficient, including reducing the size of the garbled table to just two ciphertexts per gate (Section 4.1.3) and enabling XOR gates to be computed without encryption (Section 4.1.2).

3.2 Goldreich-Micali-Wigderson (GMW) Protocol

As noted before, computation under encryption can be naturally viewed as operating on secret-shared data. In Yao's GC, the secret sharing of the active wire value is done by having one player (generator) hold two possible wire labels w_i^0, w_i^1, and the other player (evaluator) hold the active label w_i^b. In the GMW protocol (Goldreich *et al.*, 1987; Goldreich, 2004), the secret-sharing of the wire value is more direct: the players hold additive shares of the active wire value.

The GMW protocol (or just "GMW") naturally generalizes to more than two parties, unlike Yao's GC, which requires novel techniques to generalize to more than two parties (see Section 3.5).

PARAMETERS:
Boolean circuit C implementing function $\mathcal{F}$, security parameter κ.

GC GENERATION:

1. *Wire Label Generation*. For each wire w_i of C, randomly choose wire labels,
$$w_i^b = (k_i^b \in_R \{0,1\}^\kappa, p_i^b \in_R \{0,1\}),$$
such that $p_i^b = 1 - p_i^{1-b}$.

2. *Garbled Circuit Construction*. For each gate G_i of C in topological order:

   (a) Assume G_i is a 2-input Boolean gate implementing function g: $w_c = g(w_a, w_b)$, where input labels are $w_a^0 = (k_a^0, p_a^0), w_a^1 = (k_a^1, p_a^1), w_b^0 = (k_b^0, p_b^0), w_b^1 = (k_b^1, p_b^1)$, and the output labels are $w_c^0 = (k_c^0, p_c^0), w_c^1 = (k_c^1, p_c^1)$.

   (b) Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set
   $$e_{v_a,v_b} = H(k_a^{v_a} \,||\, k_b^{v_b} \,||\, i) \oplus w_c^{g_i(v_a,v_b)}$$
   Sort entries e in the table by the input pointers, placing entry e_{v_a,v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$.

3. *Output Decoding Table*. For each circuit-output wire w_i (the output of gate G_j) with labels $w_i^0 = (k_i^0, p_i^0), w_i^1 = (k_i^1, p_i^1)$, create garbled output table for both possible wire values $v \in \{0,1\}$. Set
$$e_v = H(k_i^v \,||\, \text{“out”} \,||\, j) \oplus v$$
(Because we are xor-ing with a single bit, we just use the lowest bit of the output of H for generating the above e_v.) Sort entries e in the table by the input pointers, placing entry e_v in position p_i^v. (There is no conflict, since $p_i^1 = p_i^0 \oplus 1$.)

Figure 3.1: Yao's Garbled Circuit protocol: GC generation

PARAMETERS: Parties P_1 and P_2 with inputs $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$ respectively. Boolean circuit C implementing function $\mathcal{F}$.

PROTOCOL:

1. P_1 plays the role of GC generator and runs the algorithm of Figure 3.1. P_1 then sends the obtained GC $\widehat{C}$ (including the output decoding table) to P_2.

2. P_1 sends to P_2 active wire labels for the wires on which P_1 provides input.

3. For each wire w_i on which P_2 provides input, P_1 and P_2 execute an Oblivious Transfer (OT) where P_1 plays the role of the Sender, and P_2 plays the role of the Receiver:

   (a) P_1's two input secrets are the two labels for the wire, and P_2's choice-bit input is its input on that wire.

   (b) Upon completion of the OT, P_2 receives active wire label on the wire.

4. P_2 evaluates received $\widehat{C}$ gate-by-gate, starting with the active labels on the input wires.

   (a) For gate G_i with garbled table $T = (e_{0,0}, ...e_{1,1})$ and active input labels $w_a = (k_a, p_a), w_b = (k_b, p_b)$, P_2 computes active output label $w_c = (k_c, p_c)$:

$$w_c = H(k_a \mid\mid k_b \mid\mid i) \oplus e_{p_a,p_b}$$

5. Obtaining output using output decoding tables. Once all gates of $\widehat{C}$ are evaluated, using "out" for the second key to decode the final output gates, P_2 obtains the final output labels which are equal to the plaintext output of the computation. P_2 sends the obtained output to P_1, and they both output it.

Figure 3.2: Yao's Garbled Circuit Protocol

3.2.1 GMW Intuition

The GMW protocol can work both on Boolean and arithmetic circuits. We present the two-party Boolean version first, and then briefly explain how the protocol can be generalized to more than two parties. As with Yao's protocol, we assume players P_1 with input x and P_2 with input y have agreed on the Boolean circuit C representing the computed function $\mathcal{F}(x, y)$.

The GMW protocol proceeds as follows. For each input bit $x_i \in \{0, 1\}$ of $x \in \{0, 1\}^n$, P_1 generates a random bit $r_i \in_R \{0, 1\}$ and sends all r_i to P_2. Next, P_1 obtains a secret sharing of each x_i among P_1 and P_2 by setting its share to be $x_i \oplus r_i$. Symmetrically, P_2 generates random bit masks for its inputs y_i and sends the masks to P_1, secret sharing its input similarly.

P_1 and P_2 proceed in evaluating C gate by gate. Consider gate G with input wires w_i and w_j and output wire w_k. The input wires are split into two shares, such that $s_x^1 \oplus s_x^2 = w_x$. Let P_1 holds shares s_i^1 and s_j^1 on w_i and w_j, and P_2 hold shares s_i^2 and s_j^2 on the two wires. Without loss of generality, assume C consists of NOT, XOR and AND gates.

Both NOT and XOR gates can be evaluated without any interaction. A NOT gate is evaluated by P_1 flipping its share of the wire value, which flips the shared wire value. An XOR gate on wires w_i and w_j is evaluated by players xor-ing the shares they already hold. That is, P_1 computes its output share as $s_k^1 = s_i^1 \oplus s_j^1$, and P_2 correspondingly computes its output share as $s_k^2 = s_i^2 \oplus s_j^2$. The computed shares, s_k^1, s_k^2, indeed are shares of the active output value: $s_k^1 \oplus s_k^2 = (s_i^1 \oplus s_j^1) \oplus (s_i^2 \oplus s_j^2) = (s_i^1 \oplus s_i^2) \oplus (s_j^1 \oplus s_j^2) = v_1 \oplus v_2$.

Evaluating an AND gate requires interaction and uses 1-out-of-4 OT a basic primitive. From the point of view of P_1, its shares s_i^1, s_j^1 are fixed, and P_2 has two Boolean input shares, which means there are four possible options for P_2. If P_1 knew P_2's shares, then evaluating the gate under encryption would be trivial: P_1 can just reconstruct the active input values, compute the active output value and secret-share it with P_2. While P_1 cannot do that, it can do the next best thing: prepare such a secret share for *each* of P_2's possible inputs, and run 1-out-of-4 OT to transfer the corresponding share. Specifically, let

$$S = S_{s_i^1, s_j^1}(s_i^2, s_j^2) = (s_i^1 \oplus s_i^2) \wedge (s_j^1 \oplus s_j^2)$$

be the function computing the gate output value from the shared secrets on the two input wires. P_1 chooses a random mask bit $r \in_R \{0, 1\}$ and prepares a

table of OT secrets:

$$T_G = \begin{pmatrix} r \oplus S(0,0) \\ r \oplus S(0,1) \\ r \oplus S(1,0) \\ r \oplus S(1,1) \end{pmatrix}$$

Then P_1 and P_2 run an 1-out-of-4 OT protocol, where P_1 plays the role of the sender, and P_2 plays the role of the receiver. P_1 uses table rows as each of the four input secrets, and P_2 uses its two bit shares as the selection to choose the corresponding row. P_1 keeps r as its share of the gate output wire value, and P_2 uses the value it receives from the OT execution.

Because of the way the OT inputs are constructed, the players obtain a secret sharing of the gate output wire. At the same time, it is intuitively clear that the players haven't learned anything about the other player's inputs or the intermediate values of the computation. This is because effectively only P_2 receives messages, and by the OT guarantee, it learns nothing about the three OT secrets it did not select. The only thing it learns is its OT output, which is its share of a random sharing of the output value and therefore leaks no information about the plaintext value on that wire. Likewise, P_1 learns nothing about the selection of P_2.

After evaluating all gates, players reveal to each other the shares of the output wires to obtain the output of the computation.

Generalization to more than two parties. We now sketch how to generalize this to the setting where n players $\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n$ evaluate a boolean circuit $\mathcal{F}$. As before, player P_j secret-shares its input by choosing $\forall i \neq j, r_i \in_R \{0,1\}$, and sending r_i to each P_i. The parties $\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n$ proceed by evaluating C gate-by-gate. They evaluate each gate G as follows:

- For an XOR gate, the players locally add their shares. Like the two-party case, no interaction is required and correctness and security are assured.
- For an AND gate $c = a \wedge b$, let $a_1, \ldots, a_n, b_1, \ldots, b_n$ denote the shares

of a, b respectively held by the players. Consider the identity

$$c = a \wedge b = (a_1 \oplus \cdots \oplus a_n) \wedge (b_1 \oplus \cdots \oplus b_n)$$
$$= \left(\bigoplus_{i=1}^{n} a_i \wedge b_i \right) \oplus \left(\bigoplus_{i \neq j} a_i \wedge b_j \right)$$

Each player P_j computes $a_j \wedge b_j$ locally to obtain a sharing of $\bigoplus_{i=1}^{n} a_i \wedge b_i$. Further, each pair of players $\mathsf{P}_i, \mathsf{P}_j$ jointly computes the shares of $a_i \wedge b_j$ as described above in the two-party GMW. Finally, each player outputs the XOR of all obtained shares as the sharing of the result $a \wedge b$.

3.3 BGW protocol

One of the first multi-party protocols for secure computation is due to Ben-Or, Goldwasser, and Wigderson (Ben-Or *et al.*, 1988), and is known as the "BGW" protocol. Another somewhat similar protocol of Chaum, Crépau, and Damgård was published concurrently (Chaum *et al.*, 1988) with BGW, and the two protocols are often considered together. For concreteness, we present here the BGW protocol for n parties, which is somewhat simpler.

The BGW protocol can be used to evaluate an arithmetic circuit over a field $\mathbb{F}$, consisting of addition, multiplication, and multiplication-by-constant gates. The protocol is heavily based on Shamir secret sharing (Shamir, 1979), and it uses the fact that Shamir secret shares are homomorphic in a special way—the underlying shared value can be manipulated obliviously, by suitable manipulations to the individual shares.

For $v \in \mathbb{F}$ we write $[v]$ to denote that the parties hold Shamir secret shares of a value v. More specifically, a dealer chooses a random polynomial p of degree at most t, such that $p(0) = v$. Each party P_i then holds value $p(i)$ as their share. We refer to t as the *threshold* of the sharing, so that any collection of t shares reveals no information about v.

The invariant of the BGW protocol is that for every wire w in the arithmetic circuit, the parties hold a secret-sharing $[v_w]$ of the value v_w on that wire. Next, we sketch the protocol with a focus on maintaining this invariant.

Input wires. For an input wire belonging to party P_i, that party knows the value v on that wire in the clear, and distributes shares of $[v]$ to all the parties.

Addition gate. Consider an addition gate, with input wires α, β and output wire γ. The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of $[v_\alpha + v_\beta]$. Suppose the incoming sharings correspond to polynomials p_α and p_β, respectively. If each party P_i locally adds their shares $p_\alpha(i) + p_\beta(i)$, then the result is that each party holds a point on the polynomial $p_\gamma(x) \stackrel{\text{def}}{=} p_\alpha(x) + p_\beta(x)$. Since p_γ also has degree at most t, these new values comprise a valid sharing $p_\gamma(0) = p_\alpha(0) + p_\beta(0) = v_\alpha + v_\beta$.

Note that addition gates require no communication among the parties. All steps are local computation. The same idea works to multiply a secret-shared value by a public constant — each party simply locally multiplies their share by the constant.

Multiplication gate. Consider a multiplication gate, with input wires α, β and output wire γ. The parties collectively hold sharings of incoming wires $[v_\alpha]$ and $[v_\beta]$, and the goal is to obtain a sharing of the product $[v_\alpha \cdot v_\beta]$. As above, the parties can locally multiply their individual shares, resulting in each party holding a point on the polynomial $q(x) = p_\alpha(x) \cdot p_\beta(x)$. However, in this case the resulting polynomial may have degree as high as $2t$ which is too high.

In order to fix the excessive degree of this secret sharing, the parties engage in a degree-reduction step. Each party P_i holds a value $q(i)$, where q is a polynomial of degree at most $2t$. The goal is to obtain a valid secret-sharing of $q(0)$, but with correct threshold.

The main observation is that $q(0)$ can be written as a linear function of the party's shares. In particular,

$$q(0) = \sum_{i=1}^{2t+1} \lambda_i q(i)$$

where the λ_i terms are the appropriate Lagrange coefficients. Hence the degree-reduction step works as follows:

1. Each party[3] P_i generates and distributes a threshold-t sharing of $[q(i)]$. To simplify the notation, we do not give names to the polynomials that underly these shares. However, it is important to keep in mind that each party P_i chooses a polynomial of degree at most t whose constant coefficient is $q(i)$.

[3] Technically, only $2t + 1$ parties need to do this.

2. The parties compute $[q(0)] = \sum_{i=1}^{2t+1} \lambda_i [q(i)]$, using local computations. Note that the expression is in terms of addition and multiplication-by-constant applied to secret-shared values.

Since the values $[q(i)]$ were shared with threshold t, the final sharing of $[q(0)]$ also has threshold t, as desired.

Note that multiplication gates in the BGW protocol require communication/interaction, in the form of parties sending shares of $[q(i)]$. Note also that we require $2t + 1 \leq n$, since otherwise the n parties do not collectively have enough information to determine the value $q(0)$, as q may have degree $2t$. For that reason, the BGW protocol is secure against t corrupt parties, for $2t < n$ (i.e., an honest majority).

Output wires. For an output wire α, the parties will eventually hold shares of the value $[v_\alpha]$ on that wire. Each party can simply broadcast its share of this value, so that all parties can learn v_α.

3.4 MPC From Preprocessed Multiplication Triples

A convenient paradigm for constructing MPC protocols is to split the problem into a *pre-processing* phase (before the parties' inputs are known) and an *online* phase (after the inputs are chosen). The pre-processing phase can produce correlated values for the parties, which they can later "consume" in the online phase. This paradigm is also used in some of the leading malicious-secure MPC protocols discussed in Chapter 6.

Intuition. To get an idea of how to defer some of the protocol effort to the pre-processing phase, recall the BGW protocol. The only real cost in the protocol is the communication that is required for every multiplication gate. However, it is not obvious how to move any of the related costs to a pre-processing phase, since the costs are due to manipulations of secret values that can only be determined in the online phase (i.e., they are based on the circuit inputs). Nonetheless, Beaver (1992) showed a clever way to move the majority of the communication to the pre-processing phase.

A *Beaver triple* (or *multiplication triple*) refers to a triple of secret-shared values $[a], [b], [c]$ where a and b are randomly chosen from the appropriate

field, and $c = ab$. In an offline phase, such Beaver triples can be generated in a variety of ways, such as by simply running the BGW multiplication subprotocol on random inputs. One Beaver triple is then "consumed" for each multiplication gate in the eventual protocol.

Consider a multiplication gate with input wires α, β. The parties hold secret sharings of $[v_\alpha]$ and $[v_\beta]$. To carry out the multiplication of v_α and v_β using a Beaver triple $[a], [b], [c]$, the parties do the following:

1. Using local computation, compute $[v_\alpha - a]$ and publicly open $d = v_\alpha - a$ (i.e., all parties announce their shares). While this value depends on the secret value v_α, it is masked by the random value a and therefore reveals no information about v_α.[4]

2. Using local computation, compute $[v_\beta - b]$ and publicly open $e = v_\beta - b$.

3. Observe the following identity:

$$\begin{aligned} v_\alpha v_\beta &= (v_\alpha - a + a)(v_\beta - b + b) \\ &= (d + a)(e + b) \\ &= de + db + ae + ab \\ &= de + db + ae + c \end{aligned}$$

   Since d and e are public, and the parties hold sharings of $[a], [b], [c]$, they can compute a sharing of $[v_\alpha v_\beta]$ by local computation only:

$$[v_\alpha v_\beta] = de + d[b] + e[a] + [c]$$

Using this technique, a multiplication can be performed using only two openings plus local computation. Overall, each party must broadcast two field elements per multiplication, compared to n field elements (across private channels) in the plain BGW protocol. While this comparison ignores the cost of generating the Beaver triples in the first place, there are methods for generating triples in a batch where the amortized cost of each triple is a constant number of field elements per party (Beerliová-Trubíniová and Hirt, 2008).

[4] Since a is used as essentially a one-time pad (and b similarly below), this triple $[a], [b], [c]$ cannot be reused again in a different multiplication gate.

Abstraction. While the BGW protocol (specifically, its degree-reduction step) deals with the details of Shamir secret shares, the Beaver-triples approach conveniently abstracts these away. In fact, it works as long as the parties have an abstract "sharing mechanism" $[v]$ with the following properties:

- *Additive homomorphism*: Given $[x]$ and $[y]$ and a public value z, parties can obtain any of $[x + y]$, $[x + z]$, $[xz]$, without interaction.
- *Opening*: Given $[x]$, parties can choose to reveal x to all parties.
- *Privacy*: An adversary (from whatever class of adversaries is being considered) can get no information about x from $[x]$.
- *Beaver triples*: For each multiplication gate, the parties have a random triple $[a], [b], [c]$ where $c = ab$.
- *Random input gadgets*: For each input wire belonging to party P_i, the parties have a random $[r]$, where r is known only to P_i. During the protocol, when P_i chooses its input value x for this wire, it can announce $\delta = x - r$ to all parties (leaking nothing about x), and they can locally compute $[x] = [r] + \delta$ from the homomorphic properties.

As long as these properties are true of an abstract sharing mechanism, the Beaver-triples approach is secure. In fact, the paradigm is also secure in the presence of *malicious* adversaries, as long as the opening and privacy properties of the sharing mechanism hold against such adversaries. Specifically, a malicious adversary cannot falsify the opening of a shared value. We use this fact later in Section 6.6.

Instantiations. Clearly Shamir secret sharing gives rise to an abstract sharing scheme $[\cdot]$ that satisfies the above properties with respect to adversaries who corrupt at most $t < n/2$ parties.

Another suitable method of sharing is simple additive sharing over a field $\mathbb{F}$. In additive sharing, $[v]$ signifies that each party P_i holds v_i where $\sum_{i=1}^{n} v_i = v$. This mechanism satisfies the appropriate homomorphic properties, and is secure against $n - 1$ corrupt parties. When using $\mathbb{F} = \{0, 1\}$, we obtain an offline-online variant of the GMW protocol (since the field operations in this case correspond to AND and XOR). Of course, an arbitrary $\mathbb{F}$ is possible as well, leading to a version of GMW for arithmetic circuits.

3.5 Constant-Round Multi-Party Computation: BMR

After Yao's (two-party) GC protocol was proposed, several *multi-party* protocols appeared, including Goldreich-Micali-Wigderson (GMW) (Goldreich, 2004; Goldreich *et al.*, 1987), presented in detail above in Section 3.2, Ben Or-Goldwasser-Wigderson (BGW) (Ben-Or *et al.*, 1988), Chaum-Crepeau-Damgård (CCD) (Chaum *et al.*, 1988). All of these protocols have a number of rounds linear in the depth of the circuit C computing $\mathcal{F}$. The Beaver-Micali-Rogaway (BMR) protocol (Beaver *et al.*, 1990) runs in a constant (in the depth of the circuit C) number of rounds, while achieving security against any $t < n$ number of corruptions among the n participating parties.

3.5.1 BMR Intuition

The BMR protocols adapt the main idea of Yao's GC to a multi-party setting. GC is chosen as a starting point due to its round-efficiency. However, a naïve attempt to port the GC protocol from the 2PC into the MPC setting gets stuck at the stage of sending the generated GC to the evaluators. Indeed, the circuit generator knows all the secrets (wire label correspondences), and if it colludes with any of the evaluators, the two colluding parties can learn the intermediate wire values and violate the security guarantees of the protocol.

The basic BMR idea is to perform a *distributed* GC generation, so that no single party (or even a proper subset of all parties) knows the GC generation secrets – the label assignment and correspondence. This GC generation can be done *in parallel* for all gates using MPC. This is possible by first generating (in parallel) all wire labels independently, and then independently and in parallel generating garbled gate tables. Because of parallel processing for all gates/wires, the GC generation is *independent* of the depth of the computed circuit C. As a result, the GC generation circuit C_{GEN} is constant-depth for all computed circuits C (once the security parameter κ is fixed). Even if the parties perform MPC evaluation of C_{GEN} that depends on the depth of C_{GEN}, the overall BMR protocol will still have constant rounds overall.

The MPC output, the GC produced by securely evaluating C_{GEN}, may be delivered to a designated player, say P_1, who will then evaluate it similarly to Yao's GC. The final technicality here is how to deliver the active input labels to P_1. There are several ways how this may be achieved, depending on how

exactly the MPC GC generation proceeded. Perhaps, it is conceptually simplest to view this as part of the GC generation computation.

In concrete terms, the above approach is not appealing due to potentially high costs of distributed generation of encryption tables, requiring the garbled row encryption function (instantiated as a PRF or hash function) evaluation inside MPC. Several protocols were proposed, which allow the PRF/hash evaluation to be extracted from inside the MPC and instead be done locally by the parties while providing the output of PRF/hash into the MPC. The underlying idea of such an approach is to assign different portions of each label to be generated by different players. That is, a wire w_a's labels w_a^v are a concatenation of sublabels $w_{a,j}^v$ each generated by P_j. Then, for a gate G_i with input labels $w_a^{v_a}, w_b^{v_b}$ and the output label $w_c^{v_c}$, the garbled row corresponding to input values v_a, v_b and output value v_c can simply be:

$$e_{v_a,v_b} = w_c^{v_c} \bigoplus_{j=1..n} (F(i, w_{a,j}^{v_a}) \oplus F(i, w_{b,j}^{v_b})), \tag{3.1}$$

where $F : \{0,1\}^\kappa \mapsto \{0,1\}^{n\cdot\kappa}$ is a PRG extending κ bits into $n \cdot \kappa$ bits.

The generation of the garbled table row is almost entirely done locally by each party. Each P_j computes $F(i, w_{a,j}^{v_a}) \oplus F(i, w_{b,j}^{v_b})$ and submits it to the MPC, which simply xors all the values to produce the garbled row.

However, we are not quite done. Recall that the GC evaluator P_1 will reconstruct active labels. A careful reader would notice that knowledge of its own contributed sublabel will allow it to identify which plaintext value the active label corresponds to, violating the security guarantee.

The solution is for each player P_j to add a "flip" bit $f_{a,j}$ to each wire w_a. The xor of the n flip bits, $f_a = \bigoplus_{j=1..n} f_{a,j}$, determines which plaintext bit corresponds to the wire label w_a^v. The flip bits will be an additional input into the garbling MPC. Now, with the addition of the flip bits, no subset of players will know the wire flip bit, and hence even the recognition and matching of the sublabel will not allow the evaluator to match the label to plaintext value, or to compute the inactive label in full.

We sketch a concrete example of an efficient BMR garbling in Figure 3.3; BMR evaluation is straightforward based on the garbling technique.

PARAMETERS: Boolean circuit C implementing function $\mathcal{F}$.
Let $F : \{0,1\}^\kappa \mapsto \{0,1\}^{n\cdot\kappa+1}$ be a PRG.

PLAYERS: P_1, P_2, ..., P_n with inputs $x_1, ..., x_n \in \{0,1\}^k$.

GC GENERATION:

1. For each wire w_i of C, each P_j randomly chooses wire sublabels, $w^b_{i,j} = (k^b_{i,j}, p^b_{i,j}) \in_R \{0,1\}^{\kappa+1}$, such that $p^b_{i,j} = 1 - p^{1-b}_{i,j}$, and flip-bit shares $f_{i,j} \in_R \{0,1\}$. For each wire w_i, P_j locally computes its underlying-MPC input,

$$I_{i,j} = (F(w^0_{i,j}), F(w^1_{i,j}), p^0_{i,j}, f_{i,j}).$$

2. For each gate G_i of C in parallel, all players participate in n-party MPC to compute the garbled table, taking as input all players' inputs $x_1, ..., x_n$ as well as pre-computed values $I_{i,j}$, by evaluating the following function:

   1. Assume G_i is a 2-input Boolean gate implementing function g, with input wires w_a, w_b and output wire w_c.
   2. Compute pointer bits $p^0_a = \bigoplus_{j=1..n} p^0_{a,j}, p^0_b = \bigoplus_{j=1..n} p^0_{b,j}, p^0_c = \bigoplus_{j=1..n} p^0_{c,j}$, and set $p^1_a = 1 - p^0_a, p^1_b = 1 - p^0_b, p^1_c = 1 - p^0_c$. Similarly compute flip bits f_a, f_b, f_c by xor-ing the corresponding flip bit shares submitted by the parties. Amend the semantics of the wires according to the flip bits by xor-ing f_a, f_b, f_c in the label index as appropriate (included in the next steps).
   3. Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values, $v_a, v_b \in \{0,1\}$, set

$$e_{v_a,v_b} = w^{v_c \oplus f_c}_c \bigoplus_{j=1..n} (F(i, w^{v_a \oplus f_a}_{a,j}) \oplus F(i, w^{v_b \oplus f_b}_{b,j})),$$

   where $w^0_c = w^0_{c,1} \,||\, \ldots \,||\, w^0_{c,n} \,||\, p^0_c, w^1_c = w^1_{c,1} \,||\, \ldots \,||\, w^1_{c,n} \,||\, p^1_c$. Sort entries e in the table, placing entry e_{v_a,v_b} in position $(p^{v_a}_a, p^{v_b}_b)$.
   4. Output to P_1 the computed garbled tables, as well as active wire labels inputs of C, as selected by players' inputs, $x_1, ..., x_n$.

Figure 3.3: BMR Multi-Party GC Generation

3.6 Information-Theoretic Garbled Circuits

Yao's GC and the GMW protocol present two different flavors of the use of secret sharing in MPC. In this section, we discuss a third flavor, where the secrets are shared not among players, but *among wires*. This construction is also interesting because it provides information-theoretic security in the OT-hybrid setting, meaning that no computational hardness assumptions are used in the protocol beyond what is used in the underlying OT. An important practical reason to consider IT GC is that it presents a trade-off between communication bandwidth and latency: it needs to send less data than Yao GC at the cost of additional communication rounds. While most research on practical MPC focuses on low round complexity, we believe some problems which require very wide circuits, such as those that arise in machine learning, may benefit from IT GC constructions.

Information-theoretic constructions typically provide stronger security at a higher cost. Surprisingly, this is not the case here. Intuitively, higher performance is obtained because information-theoretic encryption allows the encryption of a bit to be a single bit rather than a ciphertext whose length scales with the security parameter. Further, information-theoretic encryption here is done with bitwise XOR and bit shufflings, rather than with standard primitives such as AES.

We present the Gate Evaluation Secret Sharing (GESS) scheme of Kolesnikov (2005) (Kolesnikov (2006) provides details), which is the most efficient information-theoretic analog of GC. The main result of Kolesnikov (2005) is a two-party protocol for a Boolean formula F with communication complexity $\approx \sum d_i^2$, where d_i is the depth of the i-th leaf of F.

At a high level, GESS is a secret-sharing scheme, designed to allow evaluation under encryption of a Boolean gate G. The output wire labels of G are the two secrets from which P_1 produces four secret shares, one corresponding to each of the wire labels of the two input wires. GESS guarantees that a valid combination of shares (one share per wire) can be used to reconstruct the corresponding label of the output wire. This is similar to Yao's GC, but GESS does not require the use of garbled tables, and hence can be viewed as a generalization of Yao's GC. Similarly to Yao's GC approach, the secret sharing can be applied gate-by-gate without the need to decode or reconstruct the plaintext values.

Consider a two-input Boolean gate G. Given the possible output values s_0, s_1 and the semantics of the gate G, P_1 generates input labels (sh_{10}, sh_{11}), (sh_{20}, sh_{21}), such that each possible pair of encodings $(sh_{1,i}, sh_{2,j})$ where $i, j \in \{0, 1\}$, allows reconstructing $G(i, j)$ but carries no other information. Now, if P_2 obtains shares corresponding to gate inputs, it would be able to reconstruct the label on the output wire, and nothing else.

This mostly corresponds to our intuition of secret sharing schemes. Indeed, the possible gate outputs play the role of secrets, which are shared and then reconstructed from the input wires encodings (shares).

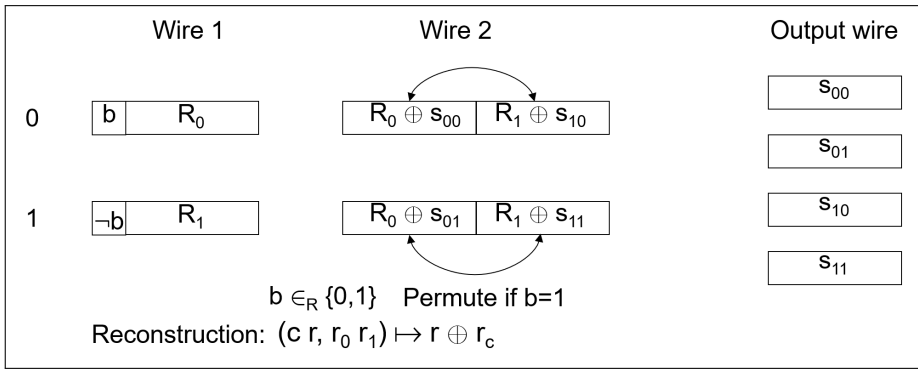
3.6.1 GESS for Two-Input Binary Gates

We present GESS for the 1-to-1 gate function $G : \{0, 1\}^2 \mapsto \{00, 01, 10, 11\}$, where $G(0, 0) = 00, G(0, 1) = 01, G(1, 0) = 10, G(1, 1) = 11$. Clearly, this is a generalization of the Boolean gate functionality $G : \{0, 1\}^2 \mapsto \{0, 1\}$.

Let the secrets domain be $\mathcal{D}_S = \{0, 1\}^n$, and four (not necessarily distinct) secrets $s_{00}, ...s_{11} \in \mathcal{D}_S$ are given. The secret s_{ij} corresponds to the value $G(i, j)$ of the output wire.

The intuition for the design of the GESS scheme is as follows (see illustration in Figure 3.4). We first randomly choose two strings $R_0, R_1 \in_R \mathcal{D}_S$ to be the shares sh_{10} and sh_{11} (corresponding to 0 and 1 of the first input wire). Now consider sh_{20}, the share corresponding to 0 of the second input wire. We want this share to produce either s_{00} (when combined with sh_{10}) or s_{10} (when combined with sh_{11}). Thus, the share sh_{20} will consist of two blocks. One, block $s_{00} \oplus R_0$, is designed to be combined with R_0 and reconstruct s_{00}. The other, $s_{10} \oplus R_1$, is designed to be combined with R_1 and reconstruct s_{10}. Share sh_{21} is constructed similarly, setting blocks to be $s_{01} \oplus R_0$ and $s_{11} \oplus R_1$.

Both leftmost blocks are designed to be combined with the same share R_0, and both rightmost blocksare designed to be combined with the same share R_1. Therefore, we append a 0 to R_0 to tell *Rec* to use the left block of the second share for reconstruction, and append a 1 to R_1 to tell *Rec* to use the right block of the second share for reconstruction. Finally, to hide information leaked by the order of blocks in shares, we randomly choose a bit b and if $b = 1$ we reverse the order of blocks in *both* shares of wire 2 and invert the appended pointer bits of the shares of wire 1. Secret reconstruction proceeds by xor-ing the wire-1 share (excluding the pointer bit) with the first or second half of the



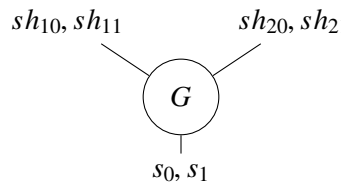


Figure 3.4: GESS for Boolean gate

wire-2 share as indexed by the pointer bit.

3.6.2 Reducing Share Growth

Note the inefficiency of the above construction, causing the shares corresponding to the second input wire be double the size of the gate's secrets. While, in some circuits we can avoid the exponential (in depth) secret growth by balancing the direction of greater growth toward more shallow parts of the circuit, a more efficient solution is desirable. We discuss only AND and OR gates, since NOT gates are implemented simply by flipping the wire label semantics by the Generator. GESS also enables XOR gates without any increase the share sizes. We defer discussion of this to Section 4.1.2, because the XOR sharing in GESS led to an important related improvement for Yao's GC.

For OR and AND gates in the above construction, either the left or the right blocks of the two shares are equal (this is because $s_{00} = s_{01}$ for the AND gate, and $s_{10} = s_{11}$ for the OR gate). We use this property to reduce the size of the shares when the secrets are of the above form. The key idea is to view the

shares of the second wire as being the same, except for one block.

Suppose each of the four secrets consists of n blocks and the secrets differ only in the j^{th} block, as follows:

$$s_{00} = (\quad t_1 \quad \ldots \quad t_{j-1} \quad t_j^{00} \quad t_{j+1} \quad \ldots \quad t_n \quad),$$
$$\ldots$$
$$s_{11} = (\quad t_1 \quad \ldots \quad t_{j-1} \quad t_j^{11} \quad t_{j+1} \quad \ldots \quad t_n \quad),$$

where $\forall i = 1..n$: $t_i, t_j^{00}, t_j^{01}, t_j^{10}, t_j^{11} \in \{0,1\}^k$ for some k. It is convenient to consider the *columns* of blocks, spanning across the shares. Every column (with the exception of the j-th) consists of four equal blocks, where the value j is private.

For simplicity, we show the main ideas by considering a special case where the four secrets consist of $n = 3$ blocks each, and $j = 2$ is the index of the column of distinct blocks. This intuition is illustrated on Figure 3.5. The scheme naturally generalizes from this intuition; Kolesnikov (2005) provides a formal presentation.

The idea is to share the secrets "column-wise", treating each of the three columns of blocks of secrets as a tuple of subsecrets and sharing this tuple separately, producing the corresponding subshares. Consider sharing column 1. All four subsecrets are equal (to t_1), and we share them trivially by setting both subshares of the first wire to a random string $R_1 \in_R \mathcal{D}_S$, and both subshares of the second wire to be $R_1 \oplus t_1$. Column 3 is shared similarly. We share column 2 as in previous construction (highlighted on the diagram), omitting the last step of appending the pointers and applying the permutation. This preliminary assignment of shares (still leaking information due to order of blocks) is shown on Figure 3.5.

Note that the reconstruction of secrets is done by xor-ing the corresponding blocks of the shares, and, importantly, the procedure is the same for both types of sharing we use. For example, given shares sh_{10} and sh_{21}, we reconstruct the secret $s_{01} = (R_1 \oplus (R_1 \oplus t_1), \quad R_2 \oplus (R_2 \oplus t_2^{01}), \quad R_3 \oplus (R_3 \oplus t_3))$.

The remaining permute-and-point step is to apply (the same) random permutation π to reorder the four columns of both shares of wire 2 and to append ($\log 4$)-bit pointers to each block of the shares of wire 1, telling the reconstructor which block of the second share to use. Note that the pointers appended to both blocks of column 1 of wire 1 are the same. The same holds

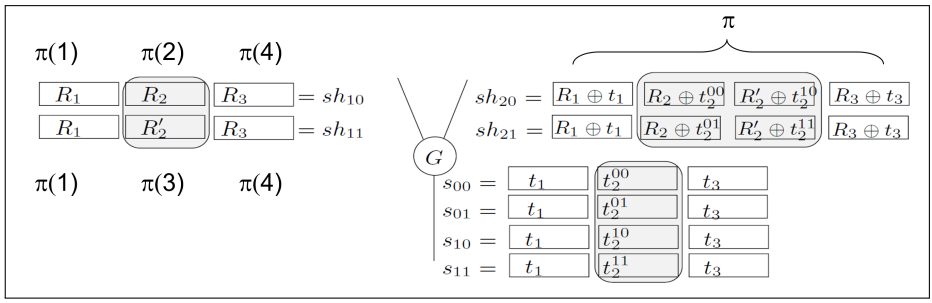


Figure 3.5: Improved GESS for Boolean gate

for column 3. Pointers appended to blocks of column 2 are different. For example, if the identity permutation was applied, then we will append "1" to both blocks R_1, "2" to R_2, "3" to R'_2, and "4" to both blocks R_3. This leads to the punchline: because G is either an OR or an AND gate, both tuples of shares maintain the property that all but one of the pairs of corresponding blocks are equal between the shares of the tuple. This allows *repeated* application (i.e., continuing sharing) of GESS for OR and AND gates.

Finally, to put it all together, we sketch the GESS-based MPC protocol. P_1 represents the function $\mathcal{F}$ as a *formula* F. Then, starting with the output wires of F and taking the plaintext output wire labels as secrets, P_1 applies GESS scheme repeatedly to all gates of the circuit, assigning the GESS shares to gates' input wires until he assigns the labels to formula inputs. Then, P_1 transfers to P_2 active labels on the input wires, and P_2 repeatedly uses GESS reconstruction procedure to obtain output labels of F.

3.7 Oblivious Transfer

Oblivious Transfer, defined in Section 2.4, is an essential building block for secure computation protocols, and an inherently asymmetric primitive. Impagliazzo and Rudich (1989) showed that a reduction from OT to a symmetric-key primitive (one-way functions, PRF) implies that P≠NP. However, as first observed by Beaver (1996), a *batched* execution of OT only needs a small number of public key operations. Beaver's construction was non-black-box in the sense that a PRF needed to be represented as a circuit and evaluated as MPC. As a consequence, Beaver's result was mainly of theoretical interest.

PARAMETERS:

1. Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$. $\mathcal{S}$ has input secrets $x_1, x_2 \in \{0,1\}^n$, and $\mathcal{R}$ has a selection bit $b \in \{0,1\}$.

PROTOCOL:

1. $\mathcal{R}$ generates a public-private key pair sk, pk, and samples a random key, pk', from the public key space. If $b = 0$, $\mathcal{R}$ sends a pair (pk, pk') to $\mathcal{S}$. Otherwise (if $b = 1$), $\mathcal{R}$ sends a pair (pk', pk) to $\mathcal{S}$.
2. $\mathcal{S}$ receives (pk_0, pk_1) and sends back to $\mathcal{R}$ two encryptions $e_0 = \mathsf{Enc}_{pk_0}(x_0), e_1 = \mathsf{Enc}_{pk_1}(x_1)$.
3. $\mathcal{R}$ receives e_0, e_1 and decrypts the ciphertext e_b using sk. $\mathcal{R}$ is unable to decrypt the second ciphertext as it does not have the corresponding secret key.

Figure 3.6: Public key-based semi-honest OT.

Ishai *et al.* (2003) changed the state of affairs dramatically by proposing an extremely efficient batched OT which only required κ of public key operations for the entire batch and two or three hashes per OT.

3.7.1 Public Key-Based OT

We start with the basic public key-based OT in the semi-honest model. The construction, presented in Figure 3.6, is very simple indeed.

The security of the construction assumes the existence of public-key encryption with the ability to sample a random public key without obtaining the corresponding secret key. The scheme is secure in the semi-honest model. The Sender $\mathcal{S}$ only sees the two public keys sent by $\mathcal{R}$, so cannot predict with probability better than $\frac{1}{2}$ which key was generated without the knowledge of the secret key. Hence, the view of $\mathcal{S}$ can be simulated simply by sending two randomly-chosen public keys.

The Receiver $\mathcal{R}$ sees two encryptions and has a secret key to decrypt only one of them. The view of $\mathcal{R}$ is also easily simulated, given $\mathcal{R}$'s input and

output. $\mathsf{Sim}_{\mathcal{S}}$ will generate the public-private key pair and a random public key, and set the simulated received ciphertexts to be 1) the encryption of the received secret under the generated keypair and 2) the encryption of zero under the randomly chosen key. The simulation goes through since the difference with the real execution is only in the second encryption, and distinguisher will not be a tell apart the encryption of zero from another value due to the encryption security guarantees. Note that this semi-honest protocol provides no security against a malicious sender—the Sender $\mathcal{S}$ can simply generate two public-private key pairs, (sk_0, pk_0) and (sk_1, pk_1) and send (pk_0, pk_1) to $\mathcal{R}$, and decrypt both received ciphertexts to learn both x_1 and x_2.

3.7.2 Public Key Operations in OT

The simple protocol in Figure 3.6 requires one public key operation for both the sender and receiver for each selection bit. As used in a Boolean circuit-based MPC protocol such as Yao's GC, it is necessary to perform an OT for each input bit of the party executing the circuit. For protocols like GMW, evaluating each AND gate requires an OT. Hence, several works have focused on reducing the number of public key operations to perform a large number of OTs.

Beaver's non-black-box construction. Beaver (1996) proposed bootstrapping Yao's GC protocol to generate a polynomial number of OTs from a small number of public key operations. As discussed in Section 3.1, the GC protocol for computing a circuit C requires m OTs, where m is the number of input bits provided by P_2. Following the OT notation, we call P_1 (the generator in GC) the sender $\mathcal{S}$, and P_2 (the evaluator in GC) the receiver $\mathcal{R}$. Let m be a desired number of OTs that will now be performed as a batch. $\mathcal{S}$'s input will be m pairs of secrets $(x_1^0, x_1^1), ..., (x_m^0, x_m^1)$, and $\mathcal{R}$'s input will be m-bit selection string $b = (b_1, ..., b_m)$.

We now construct a circuit C that implements a function F which takes only a small number of input bits from $\mathcal{R}$, but outputs the result of polynomial number of OTs to $\mathcal{R}$. The input of $\mathcal{R}$ to F will be a randomly chosen κ-bit string r. Let G be a pseudo-random generator expanding κ bits into m bits. $\mathcal{R}$ will send to $\mathcal{S}$ its input string masked with the pseudo-random string, $b \oplus G(r)$. Then, $\mathcal{S}$'s input to F will be m pairs of secrets $(x_1^0, x_1^1), \ldots, (x_m^0, x_m^1)$ as well as the m-bit string $b \oplus G(r)$. Given r, the function F computes the m-bit expansion

$G(r)$ and unmasks the input $b \oplus G(r)$, obtaining the selection string b. Then F simply outputs to $\mathcal{R}$ the corresponding secrets x_{b_i}. Only κ input bits are provided by $\mathcal{R}$, the circuit evaluator, so only a constant number of κ OTs are needed to perform m OTs.

Reducing the number of public key operations. The construction of Beaver (1996) shows a simple way to reduce the number of asymmetric operations required to perform m OTs to a fixed security parameter, but is not efficient in practice because of the need to execute a large GC. Recall, our goal is to use a small number k of base-OTs, plus only symmetric-key operations, to achieve $m \gg k$ effective OTs. Here, k is chosen depending on the computational security parameter κ; in the following we show how to choose k. Below we describe the OT extension by Ishai *et al.* (2003) that achieves m 1-out-of-2 OT of random strings, in the presence of semi-honest adversaries.

We follow the notation of Kolesnikov and Kumaresan (2013), as it explicates the coding-theoretic framework for OT extension. Suppose the receiver $\mathcal{R}$ has choice bits $r \in \{0,1\}^m$. $\mathcal{R}$ chooses two $m \times k$ matrices (m rows, k columns), T and U. Let $\boldsymbol{t}_j, \boldsymbol{u}_j \in \{0,1\}^k$ denote the j-th row of T and U, respectively. The matrices are chosen at random, so that:

$$\boldsymbol{t}_j \oplus \boldsymbol{u}_j = r_j \cdot 1^k \stackrel{\text{def}}{=} \begin{cases} 1^k & \text{if } r_j = 1 \\ 0^k & \text{if } r_j = 0 \end{cases}$$

The sender $\mathcal{S}$ chooses a random string $s \in \{0,1\}^k$. The parties engage in k instances of 1-out-of-2 string-OT, *with their roles reversed*, to transfer to sender $\mathcal{S}$ the columns of either T or U, depending on the sender's bit s_i in the string s it chose. In the i-th OT, $\mathcal{R}$ provides inputs $\boldsymbol{t}^i$ and $\boldsymbol{u}^i$, where these refer to the i-th *column* of T and U, respectively. $\mathcal{S}$ uses s_i as its choice bit and receives output $\boldsymbol{q}^i \in \{\boldsymbol{t}^i, \boldsymbol{u}^i\}$. Note that these are OTs of strings of length $m \gg k$ — the *length* of OT messages is easily extended, e.g., by encrypting and sending the two m-bit long strings, and using OT on short strings to send the right decryption key.

Now let Q denote the matrix obtained by the sender, whose columns are $\boldsymbol{q}^i$. Let $\boldsymbol{q}_j$ denote the jth row. The key observation is that

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [r_j \cdot s] = \begin{cases} \boldsymbol{t}_j & \text{if } r_j = 0 \\ \boldsymbol{t}_j \oplus s & \text{if } r_j = 1 \end{cases} \tag{3.2}$$

Let H be a Random Oracle (RO)[5]. Then $\mathcal{S}$ can compute two random strings $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$, of which $\mathcal{R}$ can compute only one, via $H(\boldsymbol{t}_j)$, of $\mathcal{R}$'s choice. Indeed, following Equation 3.2, $\boldsymbol{q}_j$ equals either $\boldsymbol{t}_j$ or $\boldsymbol{t}_j \oplus s$, depending on $\mathcal{R}$'s choice bit r_j. It is immediate then that $\boldsymbol{t}_j$ equals either $\boldsymbol{q}_j$ or $\boldsymbol{q}_j \oplus s$, depending on $\mathcal{R}$'s choice bit r_j. Note that $\mathcal{R}$ has no information about s, so intuitively it can learn only one of the two random strings $H(\boldsymbol{q}_j), H(\boldsymbol{q}_j \oplus s)$. Hence, each of the m rows of the matrix can be used to produce a single 1-out-of-2 OT of random strings.

To extend this to the more usual 1-out-of-2 OT of two given secrets s_0, s_1, we add the following step to the above. $\mathcal{S}$ now additionally encrypts the two OT secrets with the two keys $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$ and sending the two encryptions (e.g. $H(\boldsymbol{q}_j) \oplus s_0$ and $H(\boldsymbol{q}_j \oplus s) \oplus s_1$) to $\mathcal{R}$. As $\mathcal{R}$ can obtain exactly one of $H(\boldsymbol{q}_j)$ and $H(\boldsymbol{q}_j \oplus s)$, he can obtain only the corresponding secret s_i.

Coding interpretation and cheaper 1-out-of-2^ℓ OT. In IKNP, the receiver prepares secret shares of T and U such that each row of $T \oplus U$ is either all zeros or all ones. Kolesnikov and Kumaresan (2013) interpret this aspect of IKNP as a *repetition code* and suggest using other codes instead.

Consider how we might use the IKNP OT extension protocol to realize 1-out-of-2^ℓ OT. Instead of a choice bit r_i for the receiver, r_i will now be an ℓ-bit string. Let C be a linear error correcting code of dimension ℓ and codeword length k. The receiver will prepare matrices T and U so that $\boldsymbol{t}_j \oplus \boldsymbol{u}_j = C(r_j)$.

Now, generalizing Equation 3.2 the sender $\mathcal{S}$ receives

$$\boldsymbol{q}_j = \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \tag{3.3}$$

where "$\cdot$" now denotes bitwise-AND of two strings of length k. (Note that when C is a repetition code, this is exactly Equation 3.2.)

For each value $r' \in \{0,1\}^\ell$, the sender associates the secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$, which it can compute for all $r' \in \{0,1\}^\ell$. At the same time, the receiver can compute one of these values, $H(\boldsymbol{t}_j)$. Rearranging Equation 3.3, we have:

$$H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r_j) \cdot s])$$

[5] As pointed out by Ishai *et al.* (2003), it is sufficient to assume that H is a correlation-robust hash function, a weaker assumption than RO. A special assumption is required because the same s is used for every resulting OT instance.

Hence, the value that the receiver can learn is the secret value that the sender associates with the receiver's choice string $r' = r_j$.

At this point, OT of random strings is completed. For OT of chosen strings, the sender will use each $H(\boldsymbol{q}_i \oplus [C(r) \cdot s])$ as a key to encrypt the r-th OT message. The receiver will be able to decrypt only one of these encryptions, namely one corresponding to its choice string r_j.

To argue that the receiver learns *only one* string, suppose the receiver has choice bits r_j but tries to learn also the secret $H(\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s])$ corresponding to a different choice $\tilde{r}$. We observe:

$$\begin{aligned}\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s] &= \boldsymbol{t}_j \oplus [C(r_j) \cdot s] \oplus [C(\tilde{r}) \cdot s] \\ &= \boldsymbol{t}_j \oplus [(C(r_j) \oplus C(\tilde{r})) \cdot s]\end{aligned}$$

Importantly, everything in this expression is known to the receiver except for s. Now suppose the minimum distance of C is κ (the security parameter). Then $C(r_j) \oplus C(\tilde{r})$ has Hamming weight at least κ. Intuitively, the adversary would have to guess at least κ bits of the secret s in order to violate security. The protocol is secure in the RO model, and can also be proven under the weaker assumption of correlation robustness, following Ishai *et al.* (2003) and Kolesnikov and Kumaresan (2013).

Finally, we remark that the width k of the OT extension matrix is equal to the length of codewords in C. The parameter k determines the number of base OTs and the overall cost of the protocol.

The IKNP protocol sets the number of OT matrix columns to be $k = \kappa$. To achieve the same concrete security as IKNP OT, the KK13 protocol (Kolesnikov and Kumaresan, 2013) requires setting $k = 2\kappa$, to account for the larger space required by the more efficient underlying code C.

3.8 Custom Protocols

All of the secure computation protocols discussed so far in this chapter are generic circuit-based protocols. Circuit-based protocols suffer from linear bandwidth cost in the size of the circuit, which can be prohibitive for large computations. There are significant overheads with circuit-based computation on large data structures, compared to, say, a RAM (Random Access Machine) representation. In Chapter 5 we discuss approaches for incorporating sublinear data structures into generic circuit-based protocols.

Another approach is to design a customized protocol for a particular problem. This has some significant disadvantages over using a generic protocol. For one, it requires designing and proving the security of a custom protocol. It also may not integrate with generic protocols, so even if there is an efficient custom protocol for computing a particular function, privacy-preserving applications often require additional pre-processing or post-processing around that function to be useful, so it may not be possible to use a custom protocol without also developing methods for connecting it with a generic protocol. Finally, although hardening techniques are known for generic protocols (Chapter 6), it may not be possible to (efficiently) harden a customized protocol to work in a malicious security setting.

Nevertheless, several specialized problems do benefit from tailored solutions and the performance gains possible with custom protocols may be substantial. In this work we briefly review one such practically important problem: *private set intersection*.

3.8.1 Private Set Intersection (PSI)

The goal of private set intersection (PSI) is to enable a group of parties to jointly compute the intersection of their input sets, without revealing any other information about those sets (other than upper bounds on their sizes). Although protocols for PSI have been built upon generic MPC (Huang *et al.*, 2012a), more efficient custom protocols can be achieved by taking advantage of the structure of the problem.

We will present current state-of-the art two-party PSI (Kolesnikov *et al.*, 2016). It is built on the protocol of Pinkas *et al.* (2015), which heavily uses Oblivious PRF (OPRF) as a subroutine. OPRF is an MPC protocol which allows two players to evaluate a PRF F, where one of the players holds the PRF key k, and the other player holds the PRF input x, and the second player gets $F_k(x)$. We first describe how to obtain PSI from OPRF, and then we briefly discuss the OPRF construction. The improvement of Kolesnikov *et al.* (2016) is due to developing a faster OPRF.

PSI from OPRF. We now describe the Pinkas-Schneider-Segev-Zohner (PSSZ) construction (Pinkas *et al.*, 2015) building PSI from an OPRF. For concreteness, we describe the parameters used in PSSZ when the parties have

roughly the same number n of items.

The protocol relies on Cuckoo hashing (Pagh and Rodler, 2004) with 3 hash functions, which we briefly review now. To assign n items into b bins using Cuckoo hashing, first choose random functions $h_1, h_2, h_3 : \{0,1\}^* \to [b]$ and initialize empty bins $\mathcal{B}[1], \ldots, \mathcal{B}[b]$. To hash an item x, first check to see whether any of the bins $\mathcal{B}[h_1(x)]$, $\mathcal{B}[h_2(x)]$, $\mathcal{B}[h_3(x)]$ are empty. If so, then place x in one of the empty bins and terminate. Otherwise, choose a random $i \in \{1,2,3\}$, evict the item currently in $\mathcal{B}[h_i(x)]$ and replace it with x, and then recursively try to insert the evicted item. If this process does not terminate after a certain number of iterations, then the final evicted element is placed in a special bin called the *stash*.

PSSZ uses Cuckoo hashing to implement PSI. First, the parties choose 3 random hash functions h_1, h_2, h_3 suitable for 3-way Cuckoo hashing. Suppose P_1 has input set X and P_2 has input set Y, where $|X| = |Y| = n$. P_2 maps its items into $1.2n$ bins using Cuckoo hashing and a stash of size s. At this point, P_2 has at most one item per bin and at most s items in its stash. P_2 pads its input with dummy items so that each bin contains exactly one item and the stash contains exactly s items.

The parties then run $1.2n + s$ instances of an OPRF, where P_2 plays the role of receiver and uses each of its $1.2n + s$ items as input to the OPRF. Let $F(k_i, \cdot)$ denote the PRF evaluated in the i-th OPRF instance. If P_2 has mapped item y to bin i via Cuckoo hashing, then P_2 learns $F(k_i, y)$; if P_2 has mapped y to position j in the stash, then P_2 learns $F(k_{1.2n+j}, y)$.

On the other hand, P_1 can compute $F(k_i, \cdot)$ for any i. So, P_1 computes sets of candidate PRF outputs:

$$H = \{F(k_{h_i(x)}, x) \mid x \in X \text{ and } i \in \{1,2,3\}\}$$
$$S = \{F(k_{1.2n+j}, x) \mid x \in X \text{ and } j \in \{1, \ldots, s\}\}$$

P_1 randomly permutes elements of H and elements of S and sends them to P_2, who can identify the intersection of X and Y as follows. If P_2 has an item y mapped to the stash, it checks whether the associated OPRF output is present in S. If P_2 has an item y mapped to a hashing bin, it checks whether its associated OPRF output is in H.

Intuitively, the protocol is secure against a semi-honest P_2 by the PRF property. For an item $x \in X \setminus Y$, the corresponding PRF outputs $F(k_i, y)$ are

pseudorandom. Similarly, if the PRF outputs are pseudorandom even under related keys, then it is safe for the OPRF protocol to instantiate the PRF instances with related keys.

The protocol is correct as long as the PRF does not introduce any further collisions (i.e., $F(k_i, x) = F(k_{i'}, x')$ for $x \neq x'$). We must carefully set the parameters required to prevent such collisions.

More efficient OPRF from 1-out-of-∞ OT. Kolesnikov *et al.* (2016) developed an efficient OPRF construction for the PSI protocol, by pushing on the coding idea from Section 3.7.2. The main technical observation is pointing out that the code C need not have many of the properties of error-correcting codes. The resulting pseudorandom codes enable an 1-out-of-∞ OT, which can be used to produce an efficient PSI.

In particular,

1. it makes no use of decoding, thus the code does not need to be efficiently decodable, and

2. it requires only that for all possibilities r, r', the value $C(r) \oplus C(r')$ has Hamming weight at least equal to the computational security parameter κ. In fact, it is sufficient even if the Hamming distance guarantee is only probabilistic — i.e., it holds with overwhelming probability over choice of C (we discuss subtleties below).

For ease of exposition, imagine letting C be a random oracle with suitably long output. Intuitively, when C is sufficiently long, it should be hard to find a near-collision. That is, it should be hard to find values r and r' such that $C(r) \oplus C(r')$ has low (less than a computational security parameter κ) Hamming weight. A random function with output length $k = 4\kappa$ suffices to make near-collisions negligible (Kolesnikov *et al.*, 2016).

We refer to such a function C (or family of functions, in our standard-model instantiation) as a *pseudorandom code* (PRC), since its coding-theoretic properties — namely, minimum distance — hold in a cryptographic sense.

By relaxing the requirement on C from an error-correcting code to a pseudorandom code, we *remove the a-priori bound* on the size of the receiver's choice string! In essence, the receiver can use *any string* as its choice string; the sender can associate a secret value $H(\boldsymbol{q}_j \oplus [C(r') \cdot s])$ for any string r'. As

discussed above, the receiver is only able to compute $H(\boldsymbol{t}_j) = H(\boldsymbol{q}_j \oplus [C(r) \cdot s])$ — the secret corresponding to its choice string r. The property of the PRC is that, with overwhelming probability, all other values of $\boldsymbol{q}_j \oplus [C(\tilde{r}) \cdot s]$ (that a polytime player may ever ask) differ from $\boldsymbol{t}_j$ in a way that would require the receiver to guess at least κ bits of s.

Indeed, we can view the functionality achieved by the above 1-out-of-∞ OT as a kind of OPRF. Intuitively, $r \mapsto H(\boldsymbol{q} \oplus [C(r) \cdot s])$ is a function that the sender can evaluate on any input, whose outputs are pseudorandom, and which the receiver can evaluate only on its chosen input r.

The main subtleties in viewing 1-out-of-∞ OT as OPRF are:

1. the fact that the receiver learns slightly more than the output of this "PRF" — in particular, the receiver learns $\boldsymbol{t} = \boldsymbol{q} \oplus [C(r) \cdot s]$ rather than $H(\boldsymbol{t})$; and,

2. the fact that the protocol realizes many instances of this "PRF" but with related keys — s and C are shared among all instances.

Kolesnikov *et al.* (2016) show that this construction can be securely used in place of the OPRF in the PSSZ protocol, and can scale to support private intersections of sets (of any size element) with $n = 2^{20}$ over a wide area network in under 7 seconds.

Set intersection of multiple sets can be computed iteratively by computing pairwise intersections. However, extending the above 2PC PSI protocol to the multi-party setting is not immediate. Several obstacles need to be overcome, such as the fact that in 2PC computation one player learns the set intersection of the two input sets. In the multi-party setting this information must be protected. Efficient extension of the above PSI protocol to the multi-party setting was proposed by Kolesnikov *et al.* (2017a).

3.9 Further Reading

In this book, we aim to provide an easy to understand and exciting introduction to MPC, so omit a lot of formalization and proofs. Yao's GC, despite its simplicity, has several technical proof subtleties, which are first noticed and written out in the first formal account of Yao's GC (Lindell and Pinkas, 2009). The GMW protocol was introduced by Goldreich *et al.* (1987), but Goldreich

(2004) provides a cleaner and more detailed presentation. The BGW and CCD protocols were developed concurrently by Ben-Or *et al.* (1988) and Chaum *et al.* (1988). Beaver *et al.* (1990) considered constant-round multiparty protocols. A more detailed protocol presentation and discussion can be found in Phillip Rogaway's Ph.D. thesis (Rogaway, 1991).

Recently, a visual cryptography scheme for secure computation without computers was designed based on the GESS scheme (D'Arco and De Prisco, 2014; D'Arco and De Prisco, 2016). The OT extension of Ishai *et al.* (2003) is indeed one of the most important advances in MPC, and there are several extensions. Kolesnikov and Kumaresan (2013) and Kolesnikov *et al.* (2016) propose random 1-out-of-n OT and 1-out-of-∞ OT at a cost similar to that of 1-out-of-2 OT. The above schemes are in the semi-honest model; maliciously-secure OT extensions were proposed Asharov *et al.* (2015b) and Keller *et al.* (2015) (the latter is usually seen as simpler and more efficient of the two).

Custom PSI protocols have been explored in many different settings with different computation vs. communication costs and a variety of trust assumptions. Hazay and Lindell (2008) presented a simple and efficient private set intersection protocol that assumes one party would perform computations using a trusted smartcard. Kamara *et al.* (2014) present a server-aided private set intersection protocol, which, in the case of the semi-honest server, computes the private set intersection of billion-element sets in about 580 seconds while sending about 12.4 GB of data. This is an example of asymmetric trust, which we discuss further in Section 7.2.

There has been much research on custom protocols beyond PSI, but it is surprisingly rare to find custom protocols that substantially outperform fast generic MPC implementations of the same problem.