CMP_SC 8001 - Introduction to Secure Multiparty Computation

Fundamental MPC Protocols - Part 2

Wei Jiang

Department of Electrical Engineering and Computer Science

University of Missouri

Outline

Computation on Secret Shares

- All of the approaches can be viewed as a form of computing under encryption, or computing on secretly shared inputs
- For example, an encryption $Enc_k(m)$ of a message m with a key k can be seen as secret-sharing m:
  - where one share is k and the other is $Enc_k(m)$
- This chapter presents several fundamental protocols illustrating a variety of generic approaches to secure computation
- These protocols are secure in the semi-honest adversary model

Common Protocols

Protocol	# parties	# rounds	Circuit types
Yao's GC	2	constant	Boolean
GMW	many	circuit depth	Boolean or arithmetic
BGW	many	circuit depth	Boolean or arithmetic
BMR	many	constant	Boolean
GESS	2	constant	Boolean formula

Outline

Yao's Garbled Circuits (GC) Protocol

- GC is the most widely known and celebrated MPC technique
- It is usually seen as best-performing, and many of the protocols covered in the this book build on Yao's GC
- While not having the best known communication complexity, it runs in constant rounds and avoids the costly latency
- E.g., GMW whose the number of communication rounds scales with the circuit depth

Function as a Look-up Table

- To evaluate a function $\mathcal{F}(x, y)$ where party P_1 holds $x \in X$ and P_2 holds $y \in Y$, and X and Y are the respective domains for the inputs of P_1 and P_2
- Suppose the input domain is small, and we can efficiently enumerate all possible input pairs (x, y)
  - $\mathcal{F}$ 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

Evaluating a Look-up Table

- P_1 encrypts T by assigning a randomly-chosen strong key to each possible input x and y
  - for each $x \in X$ and each $y \in Y$, P_1 chooses $k_x \in_R \{0,1\}^\kappa$ and $k_y \in_R \{0,1\}^\kappa$
  - encrypting each element $T_{x,y}$ of T with both keys k_x and k_y
- P_1 sends k_x and the encrypted (and randomly permuted) table $\langle Enc_{k_x,k_y}(T_{x,y})\rangle$ to P_2

Evaluating a Look-up Table

- Using 1-out-of-$|Y|$ oblivious transfer, k_y is sent to P_2
- Using k_x and k_y, P_2 can obtain the output $\mathcal{F}$ by decrypting $T_{x,y}$
- No other information is obtained by P2:
    - He has a single pair of keys, that can only open (decrypt) a single table entry
    - 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

Point-and-Permute

- How P_2 knows which row of T to decrypt?
  - This information is sensitive since it depends on the inputs of both parties
- 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 will produce an entry which will not end with σ zeros

Point-and-Permute

- While the above approach works, it is inefficient for P_2, who expects to decrypt half of the rows of T
- A much better approach, often called point-and-permute, was introduced by Beaver et al. (1990)
- The idea is to interpret part of the key (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

Point-and-Permute

- To avoid collisions in table row allocation, P_1 must ensure that the pointer bits do not collide within the space of keys k_x or within the space of k_y
- Key size must be maintained to achieve the corresponding level of security
- Thus, the parties can append the pointer bits to the key and maintain the desired key length

Managing Look-up Table Size

- The above solution is inefficient as it scales linearly with the domain size of $\mathcal{F}$
- However, 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

Managing Look-up Table Size - Using Boolean Circuit

- 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

Managing Look-up Table Size - Using Boolean Circuit

- 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
  - These keys are referred as wire labels
  - The plaintext wire values are simply referred as wire values
- During the execution, depending on the inputs:
  - each wire is associated with a specific plaintext value and a corresponding wire label
  - which are called **active value** and **active label**
- The evaluator P_2 can know only the active labels, but not its corresponding value, and not the inactive labels

How to Garble a Circuit

- 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:

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

How to Garble a Circuit

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

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

How to Garble a Circuit

Key Obervations

- Each cell of the look-up table encrypts the label corresponding to the output computed by the gate
- 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

How to Garble a Circuit

- P_1 permutes the entries of each look-up table (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 input values to P_2
  - For input wires belonging to P_1's inputs, 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

Circuit Evaluation

- Upon receiving the input keys and garbled tables, P_2 proceeds with the evaluation
- To decrypt the correct row of each garbled gate is achieved by the point-and-permute technique
- 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
- 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 $\mathcal{F}$

Circuit Evaluation

- 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 plaintext value

Security Analysis in the Semi-Honest Model

- Here we assume the OT protocol is secure
- For P_1 is easy, the party receives no messages in the protocol
- For P_2, the party never sees both labels for the same wire
  - this is obviously true for the input wires, and
  - it holds inductively for all intermediate wires: (1) knowing only one label on each incoming wire of the gate, and (2) decrypt only 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

Simulating P_2's View

- To simulate P_2's view, the simulator Sim_{P2} 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

Yao's GC Protocol - Overview

- For simplicity of presentation, we describe the protocol variant based on Random Oracle
- 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

Yao's GC Protocol - Overview

- For each wire label, a pointer bit p_i, is added to the wire label following the point-and-permute technique
- The pointer bits leak no information due to being random
- But they allow the evaluator to determine which row in the garbled table to decrypt

Yao's GC Protocol - GC Generation

Parameters: A Boolean circuit C that implements function $\mathcal{F}$, and security parameter κ

1. **Wire Label Generation**: For each wire w_i of C, randomly choose wire labels

$$w_i^b = (k_i^b \in_R \{0,1\}^\kappa, p_i^b \in_R \{0,1\})$$

where $b \in \{0,1\}$, and $p_i^b = 1 - p_i^{1-b}$

Yao's GC Protocol - GC Generation

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_i: $w_c = g_i(w_a, w_b)$, the labels are

   $$\begin{aligned} w_a^0 &= (k_a^0, p_a^0),\ w_a^1 = (k_a^1, p_a^1) \\ w_b^0 &= (k_b^0, p_b^0),\ w_b^1 = (k_b^1, p_b^1) \\ w_c^0 &= (k_c^0, p_c^0),\ w_c^1 = (k_c^1, p_c^1) \end{aligned}$$

   (b) Create G_i's garbled table. For each of 4 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 $\langle p_a^{v_a}, p_b^{v_b} \rangle$

Yao's GC Protocol - GC Generation

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

Yao's GC Protocol

Parameters:

- Two 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 $\mathcal{F}$

1. P_1 plays the role of GC generator and runs the GC generation algorithm to obtain $\hat{C}$
2. P_1 sends $\hat{C}$ (including the output decoding table) and the active wire labels correspond to P_1's inputs

Yao's GC Protocol

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. At the end of OT, P_2 receives active wire label on the wire

Yao's GC Protocol

4. P_2 evaluates received $\hat{C}$ gate-by-gate, starting with the active labels on the input wires
   a. For gate G_i with garbled table $T = (e_{0,0}, e_{0,1}, e_{1,0}, e_{1,1})$ and active input labels $w_a = (k_a, p_a)$, $w_b = (k_b, p_b)$, P_2 computes active output label $w_c = (k_c, p_c)$:
   $$w_c = H(k_a||k_b||i) \oplus e_{p_a,p_b}$$
5. Obtaining output using output decoding tables
   a. Once all gates of $\hat{C}$ are evaluated, using "out" for the second key to decode the final output gates
   b. P_2 obtains the final output and sends it to P_1

Outline

GMW Protocol Overview

- 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 players hold additive shares of the active wire value
  - The protocol works a Boolean or an arithmetic circuit
  - It naturally generalizes to more than two parties

GMW Protocol - Secret Sharing of Inputs

- Assume P_1 with input x and P_2 with input y have agreed on the Boolean circuit C representing the computed function $\mathcal{F}(x, y)$
- 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, as P_2's share of x_i
  - P_1 sets its secret share of each x_i to $x_i \oplus r_i$
- Symmetrically, P_2 generates random bit masks for its inputs y_i and sends the masks to P_1, secret sharing its input similarly

GMW Gate Evaluation

- 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 hold 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
- Assume C consists of NOT, XOR and AND gates

GMW Gate Evaluation

- 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

    P_1: $s_k^1 = s_i^1 \oplus s_j^1$

    P_2: $s_k^2 = s_i^2 \oplus s_j^2$
- Evaluating an AND gate requires interaction and uses 1-out-of-4 OT basic primitive

GMW AND Gate Evaluation

- 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
- P_1 prepares 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 be the function computing the gate output value from the shared secrets on the two input wires:

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

GMW AND Gate Evaluation

- P_1 chooses a random mask bit $r \in_R \{0,1\}$ and prepares a table of OT secrets:

$$T_G = \begin{pmatrix} r \oplus S_{s_i^1,s_j^1}(0,0) \\ r \oplus S_{s_i^1,s_j^1}(0,1) \\ r \oplus S_{s_i^1,s_j^1}(1,0) \\ r \oplus S_{s_i^1,s_j^1}(1,1) \end{pmatrix}$$

- 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 for row selection
- 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

GMW Security

- Because of the way the OT inputs are constructed, the players obtain a secret sharing of the gate output wire
- Clearly, the players have not learned anything about the other player's inputs or the intermediate values of the computation
- OT guarantees that only P_2 receives messages, and it learns nothing about the three OT secrets
- P_2 only learns a random share of the output value and thus leaks no information about the plaintext value on that wire
- Likewise, P_1 learns nothing about the selection of P_2

Generalization to More Than Two Parties

- 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
- For an XOR and NOT gate, the parties $P_1, \ldots, P_n$ follow the steps similar to the two-party case, no interaction is required
  - A NOT gate is evaluated by a designed party, say P_1, flipping its share of the wire value
  - An XOR gate is evaluated by players xor-ing the shares they already hold

Multiparty AND Gate Evaluation

- For an AND gate $c = a \wedge b$, let $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ denote the shares of a and b respectively held by the players
- The AND gate can be formulated in terms of the shares:

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

Multiparty AND Gate Evaluation

- Each 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 parties P_i and P_j jointly computes the shares of $a_i \wedge b_j$ as described in the two-party GMW
- Finally, each party outputs the XOR of all obtained shares as the sharing of the result $a \wedge b$

Outline

Protocol Overview

- 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

Protocol Overview

- 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

Protocol Overview

- 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 $\langle i, p(i) \rangle$ 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

Gate Evaluations - Input Wires

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

Gate Evaluations - 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 (used to secret-share v_α and v_β), respectively

Gate Evaluations - Addition Gate

- 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

$$p_\gamma(x) \stackrel{\text{def}}{=} p_\alpha(x) + p_\beta(x)$$

- After addition, P_i has share $\langle i, p_\gamma(i) \rangle$
- Since $p_\gamma(x)$ also has degree at most t, these new values comprise a valid sharing $p_\gamma(0) = p_\alpha(0) + p_\beta(0) = v_\alpha + v_\beta$

Gate Evaluations - Multiplication Gate

- A multiplication gate has 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 $[v_\alpha \cdot v_\beta]$
- The parties can locally multiply their individual shares, resulting in each party holding a point on the polynomial

$$q(x) = p_\alpha(x) \cdot p_\beta(x)$$

- After multiplication, P_i has share $\langle i, q(i) \rangle$
- However, in this case the resulting polynomial may have degree as high as $2t$ which is too high

Degree Reduction for Multiplication Gate

- Each P_i holds a value $q(i)$, where $q(x)$ has degree at most $2t$
- The goal is to obtain a valid secret-sharing of $q(0)$, but with correct threshold bounded by t
- The main observation is that $q(0)$ can be written as a linear function of the party's shares:

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

where the λ_i terms are the appropriate Lagrange coefficients

Degree Reduction for Multiplication Gate

- For illustration purposes, assume $2t + 1 = n$, then all n parties need to participate in the computation

$$\lambda_i = \frac{\prod_{j \in \{1,\ldots,n\} \wedge j \neq i}(-j)}{\prod_{j \in \{1,\ldots,n\} \wedge j \neq i}(i - j)}$$

  Here we assume the parties' index are $\{1, \ldots, n\}$

- Since each party P_i knows the other parties indices, each party can locally derive $\lambda_1, \ldots, \lambda_n$

Degree Reduction for Multiplication Gate

- Each P_i randomly chooses a polynomial $\theta_{q(i)}$ of degree at most t from $\mathbb{F}$, generates and distributes secret shares of $q(i)$

$$\theta_{q(i)}(x) = a_{i,t}x^t + a_{i,t-1}x^{t-1} + \cdots + a_{i,1}x + q(i)$$

- At the end previous step, party P_i has

$$\theta_{q(1)}(i), \ldots, \theta_{q(n)}(i)$$

- The parties compute $[q_\gamma(0)] = \sum_{i=1}^{2t+1} \lambda_i \cdot [q(i)]$, using local computations; that is

$$q_\gamma(i) = \sum_{j=1}^{2t+1} \lambda_j \cdot \theta_{q(j)}(i)$$

$q(i) = \sum_{j=1}^{n} \lambda_j \theta_{q(i)}(j)$

$$\begin{aligned} q(1) &= \lambda_1 \theta_{q(1)}(1) + \lambda_2 \theta_{q(1)}(2) + \cdots + \lambda_n \theta_{q(1)}(n) \\ q(2) &= \lambda_1 \theta_{q(2)}(1) + \lambda_2 \theta_{q(2)}(2) + \cdots + \lambda_n \theta_{q(2)}(n) \\ &\vdots \\ q(n) &= \lambda_1 \theta_{q(n)}(1) + \lambda_2 \theta_{q(n)}(2) + \cdots + \lambda_n \theta_{q(n)}(n) \end{aligned}$$

$$
\begin{aligned}
q_\gamma(0) &= \sum_{i=1}^{n} \lambda_i q(i) \\
&= \lambda_1\lambda_1\theta_{q(1)}(1) + \lambda_1\lambda_2\theta_{q(1)}(2) + \cdots + \lambda_1\lambda_n\theta_{q(1)}(n) + \\
&\quad \lambda_2\lambda_1\theta_{q(2)}(1) + \lambda_2\lambda_2\theta_{q(2)}(2) + \cdots + \lambda_2\lambda_n\theta_{q(2)}(n) + \\
&\quad \vdots \\
&\quad \lambda_n\lambda_1\theta_{q(n)}(1) + \lambda_n\lambda_2\theta_{q(n)}(2) + \cdots + \lambda_n\lambda_n\theta_{q(n)}(n)
\end{aligned}
$$

$$
\begin{aligned}
q_\gamma(0) \quad = \quad & \lambda_1\lambda_1\theta_{q(1)}(1) + \lambda_2\lambda_1\theta_{q(1)}(2) + \cdots + \lambda_n\lambda_1\theta_{q(1)}(n) + \\
& \lambda_1\lambda_2\theta_{q(2)}(1) + \lambda_2\lambda_2\theta_{q(2)}(2) + \cdots + \lambda_n\lambda_2\theta_{q(2)}(n) + \\
& \qquad \vdots \\
& \lambda_1\lambda_n\theta_{q(n)}(1) + \lambda_2\lambda_n\theta_{q(n)}(2) + \cdots + \lambda_n\lambda_n\theta_{q(n)}(n)
\end{aligned}
$$

$$
q_\gamma(0) \quad = \quad \lambda_1 \sum_{j=1}^{n} \lambda_j\theta_{q(j)}(1) + \lambda_2 \sum_{j=1}^{n} \lambda_j\theta_{q(j)}(2) + \cdots + \lambda_n \sum_{j=1}^{n} \lambda_j\theta_{q(j)}(n)
$$

$$
q_\gamma(0) \quad = \quad \lambda_1 \cdot q_\gamma(1) + \lambda_2 \cdot q_\gamma(2) + \cdots + \lambda_n \cdot q_\gamma(n)
$$

Gate Evaluations - Multiplication Gate

- Since the values $[q(i)]$ were shared with threshold t, the final sharing of $[q_\gamma(0)]$ also has threshold t, as desired
- The multiplication gates in the BGW protocol require communication, in the form of parties sending shares of $[q(i)]$
- Also we require $2t + 1 \leq n$; otherwise, the n parties do not collectively have enough information to determine $q_\gamma(0)$
- Thus, the BGW protocol is secure against t corrupt parties, for $t < \frac{n}{2}$ (i.e., an honest majority)

Gate Evaluations - 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 reconstruct v_α

Preprocessing with Multiplicaiton Triples

- A convenient way for constructing MPC protocols is to split them 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

The Key Idea

- The BGW's real cost in the protocol is the communication 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

The Key Idea

- 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, Beaver triples can be generated in a variety of ways, such as by simply running the BGW multiplication protocol on random inputs
- One Beaver triple is then "consumed" for each multiplication gate in the eventual protocol
- Each triple can only be used for one multiplication

Multiplication with Beaver Triple

- 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 perform the following steps

Multiplication with Beaver Triple

1. Locally compute $[v_\alpha - a]$ and $[v_\beta - b]$. Publicly open $d = v_\alpha - a$ and $e = v_\beta - b$ (i.e., all parties announce their shares)
2. Observe the following relationship:

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

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:

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

Multiplication with Beaver Triple

- 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
- 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)

Acknowledgment

The contents of these slides are based on the following book:

- A Pragmatic Introduction to Secure Multi-Party Computation
  `https://securecomputation.org/`
- Chapter 3: Fundamental MPC Protocols