The Oblivious Transfer Protocol

How does Bob learn the wire-labels corresponding to his input?

- He cannot just ask Alice, since this would reveal his inputs

The Oblivious Transfer Protocol

How does Bob learn the wire-labels corresponding to his input?

- He cannot just ask Alice, since this would reveal his inputs

Security parameter

Alice and Bob use a protocol known as **oblivious transfer protocol**

- In the oblivious transfer protocol Alice has two messages m_0, m_1 of length $\ell(n)$

The Oblivious Transfer Protocol

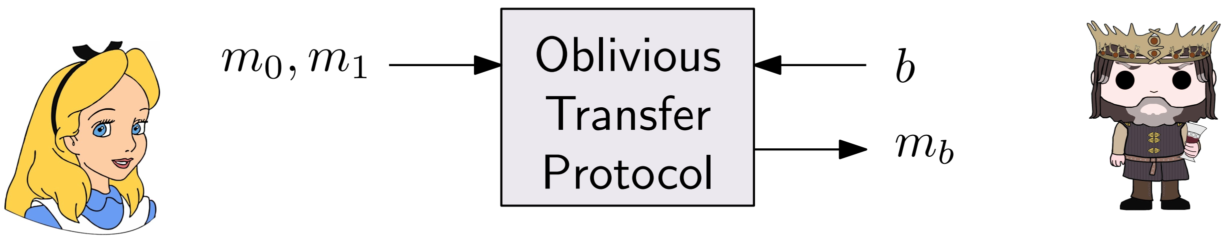
How does Bob learn the wire-labels corresponding to his input?

- He cannot just ask Alice, since this would reveal his inputs

Security parameter

Alice and Bob use a protocol known as **oblivious transfer protocol**

- In the oblivious transfer protocol Alice has two messages m_0, m_1 of length $\ell(n)$
- Bob wants to learn one of them, say m_b, without revealing which one he is interested in to Alice
- Alice wants to be sure that Bob learns exactly one of the two values



Reminder: DDH-Based Key Encapsulation Mechanism

Gen(1^n):

- Run $\mathcal{G}(1^n)$, where $\mathcal{G}$ is a group generation algorithm, to obtain (G, q, g) where G is a group of order q and $g \in G$ is a generator
- Pick some key derivation function $H : G \to \{0,1\}^{\ell(n)}$
- Choose a uniform x u.a.r. from $\{0, \ldots, q-1\}$
- Compute $h = g^x$
- Output (pk, sk) where $pk = (G, q, g, h, H)$ and $sk = (G, q, g, x, H)$.

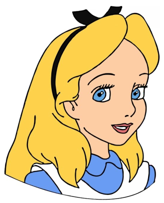
Reminder: DDH-Based Key Encapsulation Mechanism

$\mathsf{Gen}(1^n)$:

- Run $\mathcal{G}(1^n)$, where $\mathcal{G}$ is a group generation algorithm, to obtain (G, q, g) where G is a group of order q and $g \in G$ is a generator
- Pick some key derivation function $H : G \to \{0,1\}^{\ell(n)}$
- Choose a uniform x u.a.r. from $\{0, \ldots, q-1\}$
- Compute $h = g^x$
- Output (pk, sk) where $pk = (G, q, g, h, H)$ and $sk = (G, q, g, x, H)$.

$\mathsf{Encaps}_{pk}(1^n)$:

- Here $pk = (G, q, g, h, H)$
- Choose y u.a.r. from $\{0, \ldots, q-1\}$
- Output the pair (c, k) with $c = g^y$ and $k = H(h^y) = H(g^{xy})$



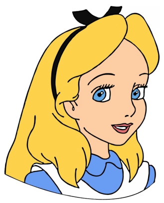
Reminder: DDH-Based Key Encapsulation Mechanism

Gen(1^n):

- Run $\mathcal{G}(1^n)$, where $\mathcal{G}$ is a group generation algorithm, to obtain (G, q, g) where G is a group of order q and $g \in G$ is a generator
- Pick some key derivation function $H : G \to \{0,1\}^{\ell(n)}$
- Choose a uniform x u.a.r. from $\{0, \ldots, q-1\}$
- Compute $h = g^x$
- Output (pk, sk) where $pk = (G, q, g, h, H)$ and $sk = (G, q, g, x, H)$.

Encaps$_{pk}(1^n)$:

- Here $pk = (G, q, g, h, H)$
- Choose y u.a.r. from $\{0, \ldots, q-1\}$
- Output the pair (c, k) with $c = g^y$ and $k = H(h^y) = H(g^{xy})$



Decaps$_{sk}(c)$:

- Here $sk = (G, q, g, x, H)$
- Output the key $H(c^x) = H(g^{xy})$

Reminder: DDH-Based KEM & Hybrid Encryption

We can build a CPA-secure PKE scheme by combining a CPA-secure KEM with an EAV-secure DEM

- We use the DDH-based KEM
- We use OTP as a DEM (for fixed-length messages)

Reminder: DDH-Based KEM & Hybrid Encryption

We can build a CPA-secure PKE scheme by combining a CPA-secure KEM with an EAV-secure DEM

- We use the DDH-based KEM
- We use OTP as a DEM (for fixed-length messages)

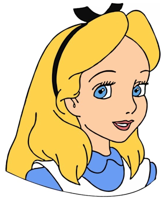
The resulting scheme is as follows:

Gen(1^n):

- Pick a group G, its order q, a generator $g \in G$, a key-derivation function $H : G \to \{0, 1\}^{\ell(n)}$
  We think of these as fixed public values agreed upon in advance between Alice and Bob.
- Pick a random $x \in G$, the **public-key** is $h = g^x$ and the **secret-key** is x

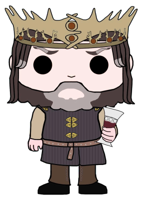
Enc$_h(m)$:

- Choose a uniform $y \in \{0, \ldots, q - 1\}$. Return the pair $c = (g^y, H(h^y) \oplus m)$



Dec$_x((c, c'))$:

- Return $H(c^x) \oplus c'$



The Oblivious Transfer Protocol

Idea:

- We pick **two** public keys h_0, h_1 for Bob
- We ensure that Bob knows the secret key x_b corresponding to **exactly** one of the public keys (of his choice)
- Alice encrypts m_0 with h_0 and m_1 with h_1
- Bob can only decrypt one of the two ciphertexts, namely the one corresponding to m_b

The Oblivious Transfer Protocol

Idea:

- We pick **two** public keys h_0, h_1 for Bob
- We ensure that Bob knows the secret key x_b corresponding to **exactly** one of the public keys (of his choice)
- Alice encrypts m_0 with h_0 and m_1 with h_1
- Bob can only decrypt one of the two ciphertexts, namely the one corresponding to m_b

How can Bob "prove" to Alice that he knows exactly one private key?

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob
- Bob picks a random private key x, and computes the two public keys:

— $h_b = g^x$. This is the public key that will be used to encrypt the message m_b wanted by Bob. The corresponding secret-key is x

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob
- Bob picks a random private key x, and computes the two public keys:
  - — $h_b = g^x$. This is the public key that will be used to encrypt the message m_b wanted by Bob. The corresponding secret-key is x
  - — $h_{1-b} = r \cdot (g^x)^{-1}$. Bob does not have the corresponding secret key

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob
- Bob picks a random private key x, and computes the two public keys:

— $h_b = g^x$. This is the public key that will be used to encrypt the message m_b wanted by Bob. The corresponding secret-key is x

— $h_{1-b} = r \cdot (g^x)^{-1}$. Bob does not have the corresponding secret key

- Bob sends h_0 and h_1 to Alice
- Alice checks that Bob "did not cheat" while computing the public keys: $h_0 \cdot h_1 = r$?

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob
- Bob picks a random private key x, and computes the two public keys:

— $h_b = g^x$. This is the public key that will be used to encrypt the message m_b wanted by Bob. The corresponding secret-key is x

— $h_{1-b} = r \cdot (g^x)^{-1}$. Bob does not have the corresponding secret key

- Bob sends h_0 and h_1 to Alice
- Alice checks that Bob "did not cheat" while computing the public keys: $h_0 \cdot h_1 = r$?
- Alice encrypts m_0 and m_1:

— Pick a uniform $y_0 \in \{0, \ldots, q-1\}$, let $c_0 = (g^{y_0}, H(h_0^{y_0}) \oplus m_0)$

— Pick a uniform $y_1 \in \{0, \ldots, q-1\}$, let $c_1 = (g^{y_1}, H(h_1^{y_1}) \oplus m_1)$

The Oblivious Transfer Protocol

- Alice picks a random group element $r \in G$ and sends it to Bob
- Bob picks a random private key x, and computes the two public keys:

— $h_b = g^x$. This is the public key that will be used to encrypt the message m_b wanted by Bob. The corresponding secret-key is x

— $h_{1-b} = r \cdot (g^x)^{-1}$. Bob does not have the corresponding secret key

- Bob sends h_0 and h_1 to Alice
- Alice checks that Bob "did not cheat" while computing the public keys: $h_0 \cdot h_1 = r$?
- Alice encrypts m_0 and m_1:

— Pick a uniform $y_0 \in \{0, \ldots, q-1\}$, let $c_0 = (g^{y_0}, H(h_0^{y_0}) \oplus m_0)$

— Pick a uniform $y_1 \in \{0, \ldots, q-1\}$, let $c_1 = (g^{y_1}, H(h_1^{y_1}) \oplus m_1)$

- Alice sends c_0 and c_1 to Bob
- Bob decrypts $c_b = (c, c')$ as $m_b = H(c^x) \oplus c'$

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}}$

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}}$

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}} = (r \cdot g^{-x})^{y_{1-b}}$ ← Requires computing the discrete logarithm of a random group element

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}} = (r \cdot g^{-x})^{y_{1-b}}$ ← Requires computing the discrete logarithm of a random group element
- Be able to evaluate $H(h_{1-b}^{y_{1-b}})$ without knowing $h_{1-b}^{y_{1-b}}$

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}} = (r \cdot g^{-x})^{y_{1-b}}$ ← Requires computing the discrete logarithm of a random group element
- Be able to evaluate $H(h_{1-b}^{y_{1-b}})$ without knowing $h_{1-b}^{y_{1-b}}$ ← Secure if H acts as a random oracle

The Oblivious Transfer Protocol: Security (informal)

Can Bob learn m_{1-b}?

- m_{1-b} was encrypted as $(g^{y_{1-b}}, H(h_{1-b}^{y_{1-b}}) \oplus m_{1-b})$

To learn m_{1-b}, Bob needs to either:

- Be able to compute $h_{1-b}^{y_{1-b}} = (r \cdot g^{-x})^{y_{1-b}}$ ← Requires computing the discrete logarithm of a random group element
- Be able to evaluate $H(h_{1-b}^{y_{1-b}})$ without knowing $h_{1-b}^{y_{1-b}}$ ← Secure if H acts as a random oracle

Secure under the Random Oracle model and the CDH assumption

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- Alice "garbles" the circuit and sends the garbled gates and the wire-labels corresponding to her input values to Bob

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- Alice "garbles" the circuit and sends the garbled gates and the wire-labels corresponding to her input values to Bob
- Bob uses the oblivious transfer protocol to learn the wire-label corresponding to each of his inputs (without Alice knowing *which* of the two labels Bob requested)

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- Alice "garbles" the circuit and sends the garbled gates and the wire-labels corresponding to her input values to Bob
- Bob uses the oblivious transfer protocol to learn the wire-label corresponding to each of his inputs (without Alice knowing *which* of the two labels Bob requested)
- Bob evaluates the garbled circuit and obtains the wire-label of the output
- Bob sends the output wire-label to Alice

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- Alice "garbles" the circuit and sends the garbled gates and the wire-labels corresponding to her input values to Bob
- Bob uses the oblivious transfer protocol to learn the wire-label corresponding to each of his inputs (without Alice knowing *which* of the two labels Bob requested)
- Bob evaluates the garbled circuit and obtains the wire-label of the output
- Bob sends the output wire-label to Alice
- Alice knows the corresponding truth value, so she learns $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$

Back to Yao's Garbled Circuits: The Overall Protocol

- Alice stars from a circuit that computes $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- Alice "garbles" the circuit and sends the garbled gates and the wire-labels corresponding to her input values to Bob
- Bob uses the oblivious transfer protocol to learn the wire-label corresponding to each of his inputs (without Alice knowing *which* of the two labels Bob requested)
- Bob evaluates the garbled circuit and obtains the wire-label of the output
- Bob sends the output wire-label to Alice
- Alice knows the corresponding truth value, so she learns $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$
- If Bob should also know the value of $f(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$, Alice shares it with Bob

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

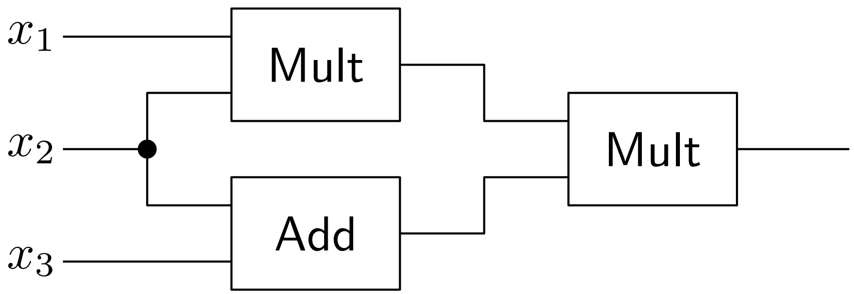
- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.

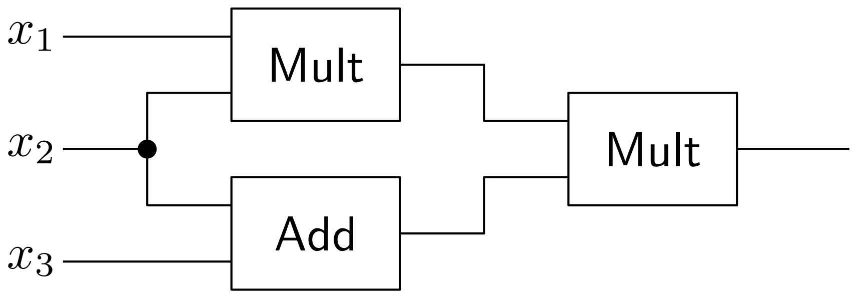


Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



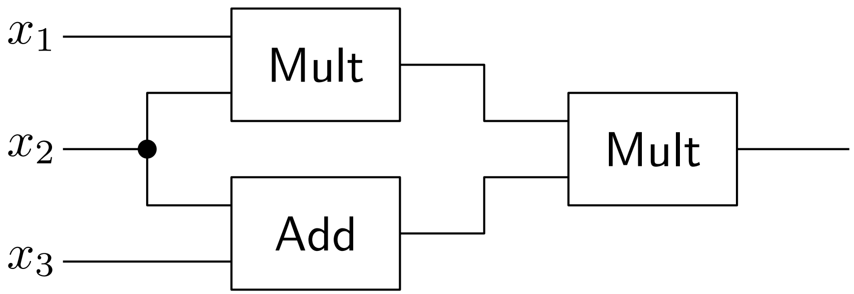
Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod p$

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod p$

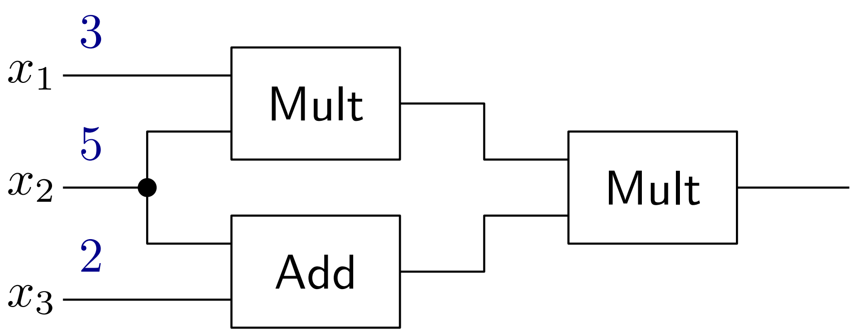
With inputs $x_1 = 3$, $x_2 = 5$, and $x_3 = 2$, and $p = 11$ it computes 6

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod{p}$

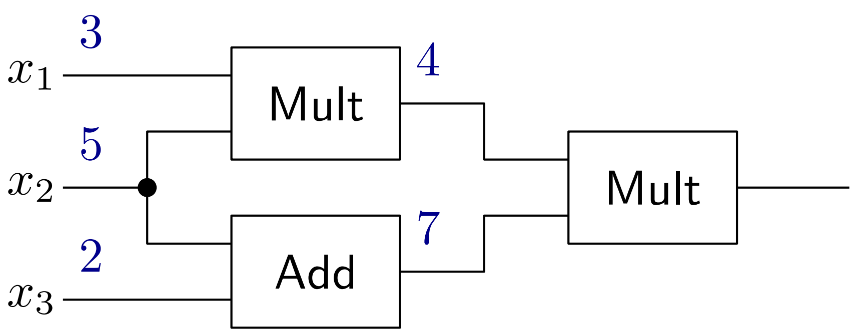
With inputs $x_1 = 3$, $x_2 = 5$, and $x_3 = 2$, and $p = 11$ it computes 6

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod{p}$

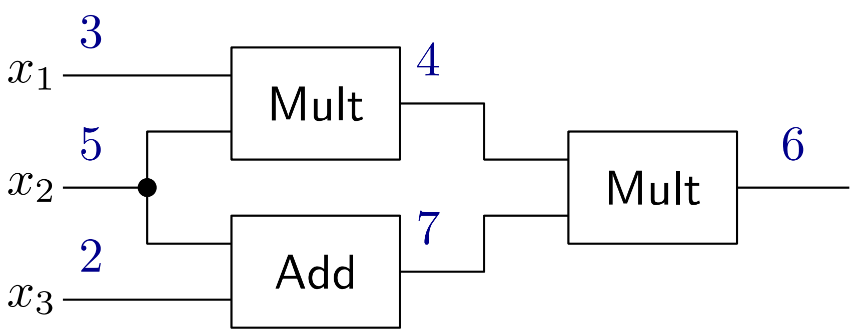
With inputs $x_1 = 3$, $x_2 = 5$, and $x_3 = 2$, and $p = 11$ it computes 6

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod{p}$

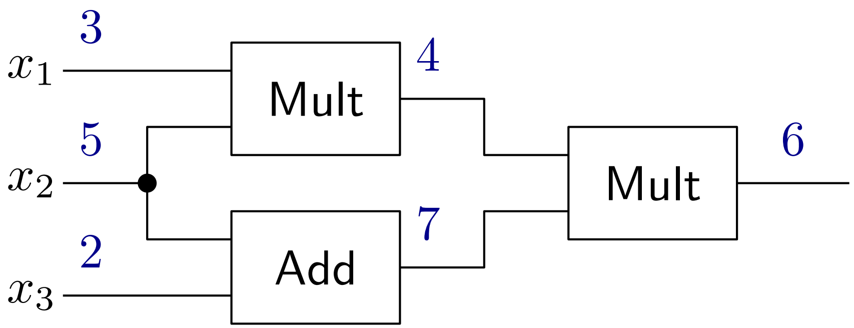
With inputs $x_1 = 3$, $x_2 = 5$, and $x_3 = 2$, and $p = 11$ it computes 6

Multiparty Computation

What if $n \geq 2$ parties want to jointly compute a function?

We consider functions $f(x_1, x_2, \ldots, x_n)$ that are computed by an **arithmetic circuit** over $\mathbb{Z}_p$, for a prime $p > n$

- The i-th input x_i is an integer in $\{0, 1, \ldots, p-1\}$ and is controlled by the i-th party
- There are two gate types: **addition gates** and **multiplication gates**, that compute the sum and product of their inputs modulo p.



Computes $(x_1 \cdot x_2) \cdot (x_2 + x_3) \pmod{p}$

With inputs $x_1 = 3$, $x_2 = 5$, and $x_3 = 2$, and $p = 11$ it computes 6

How do Boolean circuit and arithmetic circuits compare?

Arithmetic Circuits & Boolean Circuits

$\Rightarrow$ **We can simulate a Boolean circuit with an arithmetic circuit:**

Arithmetic Circuits & Boolean Circuits

$\Rightarrow$ **We can simulate a Boolean circuit with an arithmetic circuit:**

- $x_1 \wedge x_2 = x_1 \cdot x_2$
- $\neg x = 1 - x$
- $x_1 \vee x_2 = x_1 + x_2 - x_1 \cdot x_2$

Arithmetic Circuits & Boolean Circuits

$\Rightarrow$ **We can simulate a Boolean circuit with an arithmetic circuit:**

- $x_1 \wedge x_2 = x_1 \cdot x_2$
- $\neg x = 1 - x$
- $x_1 \vee x_2 = x_1 + x_2 - x_1 \cdot x_2$

$\Leftarrow$ **We can simulate an arithmetic circuit with a Boolean circuit:**

Arithmetic Circuits & Boolean Circuits

$\Rightarrow$ **We can simulate a Boolean circuit with an arithmetic circuit:**

- $x_1 \wedge x_2 = x_1 \cdot x_2$
- $\neg x = 1 - x$
- $x_1 \vee x_2 = x_1 + x_2 - x_1 \cdot x_2$

$\Leftarrow$ **We can simulate an arithmetic circuit with a Boolean circuit:**

- Replace each wire of the arithmetic circuit with $\lceil \log p \rceil$ Boolean wires
- Replace each Addition/Multiplication gate with a Boolean circuit that computes the Sum/Product of the inputs modulo p

Multiparty Computation

How do we "garble" and arithmetic circuit for multiple parties?

How do we evaluate it?

Multiparty Computation

How do we “garble” and arithmetic circuit for multiple parties?

How do we evaluate it?

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead

Multiparty Computation

How do we “garble” and arithmetic circuit for multiple parties?

How do we evaluate it?

We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead

Multiparty Computation

How do we "garble" and arithmetic circuit for multiple parties?

How do we evaluate it?

We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead
- Each party shares its input with all other parties using Shamir's k-out-of-n threshold secret sharing scheme

Multiparty Computation

How do we "garble" and arithmetic circuit for multiple parties?

How do we evaluate it?

We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead
- Each party shares its input with all other parties using Shamir's k-out-of-n threshold secret sharing scheme

k is an integer parameter that controls how resilient the protocol is to coalitions of curious parties. No group less than t parties can collude to recover the secret. The construction works for $k \leq \lceil \frac{n}{2} \rceil$

Multiparty Computation

How do we “garble” and arithmetic circuit for multiple parties?

How do we evaluate it?

We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead
- Each party shares its input with all other parties using Shamir’s k-out-of-n threshold secret sharing scheme
- Each party evaluates the arithmetic circuit: a gate takes a share for each of the two inputs and produces a share of the output

k is an integer parameter that controls how resilient the protocol is to coalitions of curious parties. No group less than t parties can collude to recover the secret. The construction works for $k \leq \lceil \frac{n}{2} \rceil$

Multiparty Computation

How do we "garble" and arithmetic circuit for multiple parties?

How do we evaluate it?

We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead
- Each party shares its input with all other parties using Shamir's k-out-of-n threshold secret sharing scheme
- Each party evaluates the arithmetic circuit: a gate takes a share for each of the two inputs and produces a share of the output
- The output of the circuit is a share of $f(x_1, \ldots, x_n)$

k is an integer parameter that controls how resilient the protocol is to coalitions of curious parties. No group less than t parties can collude to recover the secret. The construction works for $k \leq \lceil \frac{n}{2} \rceil$

Multiparty Computation

How do we “garble” and arithmetic circuit for multiple parties?

How do we evaluate it?

> We can perform computation on shares, without having to recover their secrets first!

Idea:

- Do not garble the circuit, use the **homomorphic properties** of Shamir secret sharing instead
- Each party shares its input with all other parties using Shamir's k-out-of-n threshold secret sharing scheme
- Each party evaluates the arithmetic circuit: a gate takes a share for each of the two inputs and produces a share of the output
- The output of the circuit is a share of $f(x_1, \ldots, x_n)$
- The parties combine their output shares and recover the value of $f(x_1, \ldots, x_n)$

> k is an integer parameter that controls how resilient the protocol is to coalitions of curious parties. No group less than t parties can collude to recover the secret. The construction works for $k \leq \lceil \frac{n}{2} \rceil$

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

... and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

... and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k-1$) used to share a and b

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod{p}$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k - 1$) used to share a and b
- Notice that the polynomial $f_c(x) = f_a(x) + f_b(x) \pmod{p}$ has degree at most $k - 1$ and is such that $f_c(0) = f_a(0) + f_b(0) = a + b \pmod{p}$

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k-1$) used to share a and b
- Notice that the polynomial $f_c(x) = f_a(x) + f_b(x) \pmod p$ has degree at most $k-1$ and is such that $f_c(0) = f_a(0) + f_b(0) = a + b \pmod p$
- f_c is a valid polynomial for sharing c in the Shamir's k-out-of-n threshold secret sharing scheme!

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k-1$) used to share a and b
- Notice that the polynomial $f_c(x) = f_a(x) + f_b(x) \pmod p$ has degree at most $k-1$ and is such that $f_c(0) = f_a(0) + f_b(0) = a + b \pmod p$
- f_c is a valid polynomial for sharing c in the Shamir's k-out-of-n threshold secret sharing scheme!

What is the i-th share (i, c_i) of f_c?

Addition Gates

Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k-1$) used to share a and b
- Notice that the polynomial $f_c(x) = f_a(x) + f_b(x) \pmod p$ has degree at most $k-1$ and is such that $f_c(0) = f_a(0) + f_b(0) = a + b \pmod p$
- f_c is a valid polynomial for sharing c in the Shamir's k-out-of-n threshold secret sharing scheme!

What is the i-th share (i, c_i) of f_c?

$$c_i = f_c(i) = f_a(i) + f_b(i) = a_i + b_i \pmod p$$

Addition Gates

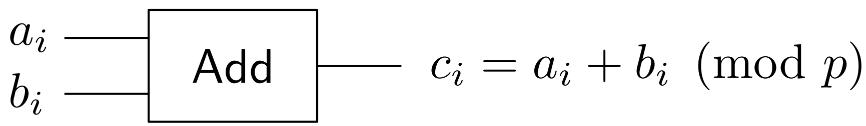
Party i has the i-th shares (i, a_i), (i, b_i) of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share (i, c_i) of the secret $c = a + b \pmod p$

- Let $f_a(x)$ and $f_b(x)$ be the polynomials (of degree at most $k - 1$) used to share a and b
- Notice that the polynomial $f_c(x) = f_a(x) + f_b(x) \pmod p$ has degree at most $k - 1$ and is such that $f_c(0) = f_a(0) + f_b(0) = a + b \pmod p$
- f_c is a valid polynomial for sharing c in the Shamir's k-out-of-n threshold secret sharing scheme!

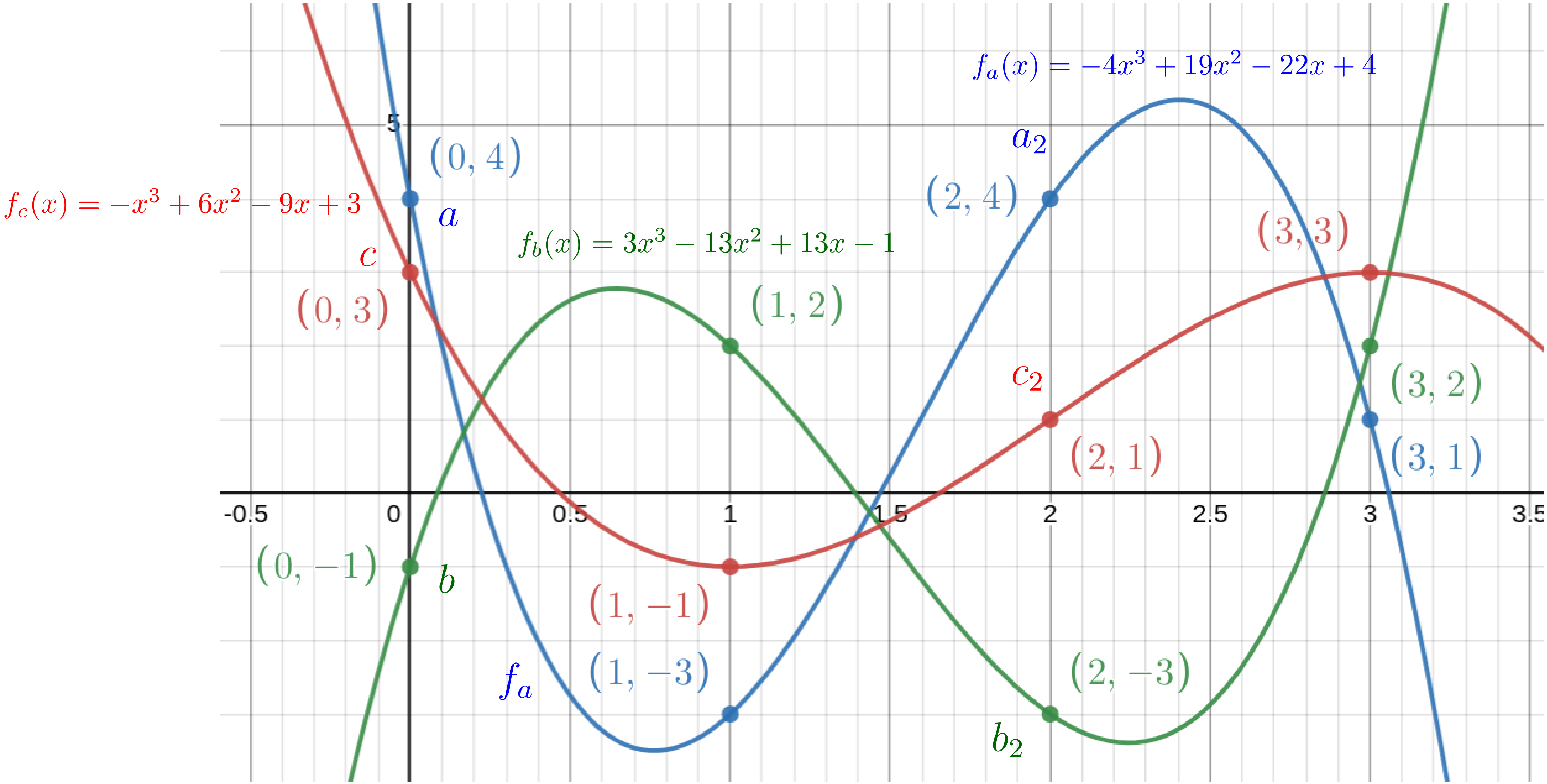
What is the i-th share (i, c_i) of f_c?

$$c_i = f_c(i) = f_a(i) + f_b(i) = a_i + b_i \pmod p$$



Addition gates do not require any special care!

Addition Gates



Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

... and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod p$

Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod{p}$

We can't just use the share (i, c_i) with $c_i = a_i \cdot b_i$ **Why?**

Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

... and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod p$

We can't just use the share (i, c_i) with $c_i = a_i \cdot b_i$ **Why?**

- We could define $f_c(x) = f_a(x) \cdot f_b(x)$, and it would satisfy $f_c(0) = a \cdot b$ ...

Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod p$

We can't just use the share (i, c_i) with $c_i = a_i \cdot b_i$ **Why?**

- We could define $f_c(x) = f_a(x) \cdot f_b(x)$, and it would satisfy $f_c(0) = a \cdot b$ ...
- Also, $c_i = a_i \cdot b_i$ would be the value of $f_c(i)$ ...

Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod p$

We can't just use the share (i, c_i) with $c_i = a_i \cdot b_i$ **Why?**

- We could define $f_c(x) = f_a(x) \cdot f_b(x)$, and it would satisfy $f_c(0) = a \cdot b$ ...
- Also, $c_i = a_i \cdot b_i$ would be the value of $f_c(i)$ ...

Problem: since f_a and f_b have deree up to $k-1$, the degree of f_c can be as high as $2(k-1)$

- After each multiplication, the number of shares needed to recover c roughly doubles

Multiplication Gates

Party i has the i-th shares a_i, b_i of two (unknown) secrets a, b, respectively, ...

...and wants to compute the i-th share c_i of the secret $c = a \cdot b \pmod{p}$

We can't just use the share (i, c_i) with $c_i = a_i \cdot b_i$ **Why?**

- We could define $f_c(x) = f_a(x) \cdot f_b(x)$, and it would satisfy $f_c(0) = a \cdot b$ ...
- Also, $c_i = a_i \cdot b_i$ would be the value of $f_c(i)$ ...

Problem: since f_a and f_b have deree up to $k - 1$, the degree of f_c can be as high as $2(k - 1)$

- After each multiplication, the number of shares needed to recover c roughly doubles

We need to use another property of interpolating polynomials...

Recombination Vectors

Lemma: Given distinct $x_1, \dots, x_n$, define $r_j = \prod_{\substack{i=1,\dots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \ldots, r_n)$ is called the **recombination vector**

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \ldots, r_n)$ is called the **recombination vector**

Proof:

From Lagrange interpolation we know that: $f(x) = \sum_{j=1}^{n} f(x_j) \prod_{\substack{i=1,\ldots,n \\ i \neq j}} (x - x_i)(x_j - x_i)^{-1}$

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \ldots, r_n)$ is called the **recombination vector**

Proof:

From Lagrange interpolation we know that: $f(x) = \sum_{j=1}^{n} \underbrace{f(x_j)}_{y_j} \prod_{\substack{i=1,\ldots,n \\ i \neq j}} (x - x_i)(x_j - x_i)^{-1}$

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \ldots, r_n)$ is called the **recombination vector**

Proof:

From Lagrange interpolation we know that: $f(x) = \sum_{j=1}^{n} \underbrace{f(x_j)}_{y_j} \underbrace{\prod_{\substack{i=1,\ldots,n \\ i \neq j}} (x - x_i)(x_j - x_i)^{-1}}_{\ell_j(x)}$

Recombination Vectors

Lemma: Given distinct $x_1, \dots, x_n$, define $r_j = \prod_{\substack{i=1,\dots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \dots, r_n)$ is called the **recombination vector**

Proof:

From Lagrange interpolation we know that: $f(x) = \sum_{j=1}^{n} \underbrace{f(x_j)}_{y_j} \underbrace{\prod_{\substack{i=1,\dots,n \\ i \neq j}} (x - x_i)(x_j - x_i)^{-1}}_{\ell_j(x)}$

$$f(0) = \sum_{j=1}^{k} f(x_j) \prod_{\substack{i=1,\dots,n \\ i \neq j}} x_i (x_i - x_j)^{-1}$$

Recombination Vectors

Lemma: Given distinct $x_1, \ldots, x_n$, define $r_j = \prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i \cdot (x_i - x_j)^{-1}$.

For any polynomial f of degree at most $n-1$:

$$f(0) = \sum_{j=1}^{k} r_j f(x_j)$$

The same holds in $\mathbb{Z}_p$!

Remark: The coefficients r_i depend **only** on the x-coordinates x_i (and **not** on the choice of f)

The vector $(r_1, r_2, \ldots, r_n)$ is called the **recombination vector**

Proof:

From Lagrange interpolation we know that: $f(x) = \sum_{j=1}^{n} \underbrace{f(x_j)}_{y_j} \underbrace{\prod_{\substack{i=1,\ldots,n \\ i \neq j}} (x - x_i)(x_j - x_i)^{-1}}_{\ell_j(x)}$

$$f(0) = \sum_{j=1}^{k} f(x_j) \underbrace{\prod_{\substack{i=1,\ldots,n \\ i \neq j}} x_i (x_i - x_j)^{-1}}_{r_j}$$

□

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k - 1$ such that $\delta_i(0) = a_i \cdot b_i \pmod{p}$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod p$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k - 1$ such that $\delta_i(0) = a_i \cdot b_i \pmod p$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n - 1$.

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod{p}$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum\limits_{i=1}^{n} r_i \cdot g(i) = \sum\limits_{i=1}^{n} r_i \cdot \delta_i(0) \pmod{p}$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod p$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum_{i=1}^{n} r_i \cdot g(i) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod p$
- Consider the polynomial h obtained as a linear combination of the δ_is using the coefficients of the recombination vector:

$$h(x) = \sum_{i=1}^{n} r_i \cdot \delta_i(x) \pmod p$$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod p$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum_{i=1}^{n} r_i \cdot g(i) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod p$
- Consider the polynomial h obtained as a linear combination of the δ_is using the coefficients of the recombination vector:

$$h(x) = \sum_{i=1}^{n} r_i \cdot \delta_i(x) \qquad h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod p$$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod{p}$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum_{i=1}^{n} r_i \cdot g(i) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod{p}$
- Consider the polynomial h obtained as a linear combination of the δ_is using the coefficients of the recombination vector:

$$h(x) = \sum_{i=1}^{n} r_i \cdot \delta_i(x) \qquad h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \;\; = c \pmod{p}$$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod{p}$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum_{i=1}^{n} r_i \cdot g(i) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod{p}$
- Consider the polynomial h obtained as a linear combination of the δ_is using the coefficients of the recombination vector:

$$h(x) = \sum_{i=1}^{n} r_i \cdot \delta_i(x) \qquad h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) = c \qquad h(i) = \sum_{j=1}^{n} r_j \cdot \delta_j(i) = c_i \pmod{p}$$

Multiplication Gates

To compute a share c_i of $c = a \cdot b$ from a_i and b_i:

- Pick a random polynomial δ_i of degree $k-1$ such that $\delta_i(0) = a_i \cdot b_i \pmod p$
- Send $\delta_i(j)$ to each other party $j \in \{1, \ldots, n\} \setminus \{i\}$
- Use the (public) recombination vector $(r_1, \ldots, r_n)$ for $\{1, \ldots, n\}$ to compute $c_i = \sum_{j=1}^{n} r_j \cdot \delta_j(i)$

Why does this work?

- Consider the polynomial $g(x) = f_a(x) \cdot f_b(x)$ of degree at most $2(k-1) \leq n-1$.
- By the previous lemma, we can write: $c = g(0) = \sum_{i=1}^{n} r_i \cdot g(i) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) \pmod p$
- Consider the polynomial h obtained as a linear combination of the δ_is using the coefficients of the recombination vector:

$$h(x) = \sum_{i=1}^{n} r_i \cdot \delta_i(x) \qquad h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) = c \qquad h(i) = \sum_{j=1}^{n} r_j \cdot \delta_j(i) = c_i \qquad \pmod p$$

- h is a polynomial of degree at most $k-1$ s.t. $h(0) = c$ and c_i is exactly the i-th share of h