CS 594 – Advanced Crypto (Spring 2026)

Alex Block

Lecture 18

March 18, 2026

The BGW Protocol

BGW Protocol

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} \colon \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.
  - Here, a CONST gate represents the map $x \mapsto \alpha \cdot x$ for fixed $\alpha \in \mathbb{F}$.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.
  - Here, a CONST gate represents the map $x \mapsto \alpha \cdot x$ for fixed $\alpha \in \mathbb{F}$.

- Party P_i holds input $x_i \in \mathbb{F}$.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.
  - Here, a CONST gate represents the map $x \mapsto \alpha \cdot x$ for fixed $\alpha \in \mathbb{F}$.

- Party P_i holds input $x_i \in \mathbb{F}$.
  - More generally, parties can hold vectors of field elements as input, but we restrict ourselves to single element inputs for ease of presentation.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.
  - Here, a CONST gate represents the map $x \mapsto \alpha \cdot x$ for fixed $\alpha \in \mathbb{F}$.

- Party P_i holds input $x_i \in \mathbb{F}$.
  - More generally, parties can hold vectors of field elements as input, but we restrict ourselves to single element inputs for ease of presentation.

Key Idea

Use Shamir Secret Sharing and run GMW.

BGW Protocol

- m-party protocol for computing any *arithmetic circuit* $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}^{s_{\mathsf{out}}}$.
  - $\mathcal{C}$ consists of MULT, ADD, and CONST gates.
  - Here, a CONST gate represents the map $x \mapsto \alpha \cdot x$ for fixed $\alpha \in \mathbb{F}$.
- Party P_i holds input $x_i \in \mathbb{F}$.
  - More generally, parties can hold vectors of field elements as input, but we restrict ourselves to single element inputs for ease of presentation.

Key Idea

Use Shamir Secret Sharing and run GMW.

- There will be some technical issues that we need to handle.

BGW Protocol: Setup

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

- For $v \in \mathbb{F}$, we let $[\![v]\!]_i$ denote the i^{th} Shamir secret share of v, held by party P_i.

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

- For $v \in \mathbb{F}$, we let $[\![v]\!]_i$ denote the i^{th} Shamir secret share of v, held by party P_i.
  - I.e., v is shared as $p(0) = v$ for random polynomial p of degree t, and $[\![v]\!]_i = p(i)$.

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

- For $v \in \mathbb{F}$, we let $[\![v]\!]_i$ denote the i^{th} Shamir secret share of v, held by party P_i.
  - I.e., v is shared as $p(0) = v$ for random polynomial p of degree t, and $[\![v]\!]_i = p(i)$.

- For all $i \in [m]$:

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

- For $v \in \mathbb{F}$, we let $[\![v]\!]_i$ denote the i^{th} Shamir secret share of v, held by party P_i.
  - I.e., v is shared as $p(0) = v$ for random polynomial p of degree t, and $[\![v]\!]_i = p(i)$.

- For all $i \in [m]$:
  - Party P_i samples a random polynomial of degree t p_{x_i} such that $p_{x_i}(0) = x_i$.

BGW Protocol: Setup

- Fix a $(t+1, m)$ Shamir Secret Sharing Scheme.
  - Any $t+1$ shares reconstruct the secret, any t shares reveal nothing.

- For $v \in \mathbb{F}$, we let $[\![v]\!]_i$ denote the i^{th} Shamir secret share of v, held by party P_i.
  - I.e., v is shared as $p(0) = v$ for random polynomial p of degree t, and $[\![v]\!]_i = p(i)$.

- For all $i \in [m]$:
  - Party P_i samples a random polynomial of degree t p_{x_i} such that $p_{x_i}(0) = x_i$.
  - P_i sends $[\![x_i]\!]_j$ to every party P_j for $j \neq i$.

$p_{x_i}(j)$

BGW Protocol: Evaluation

BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.

BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $([\![x_1]\!]_i, \ldots, [\![x_m]\!]_i)$ as the input.

BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.

BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.

BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.

P_k's View

BGW Protocol: Evaluation

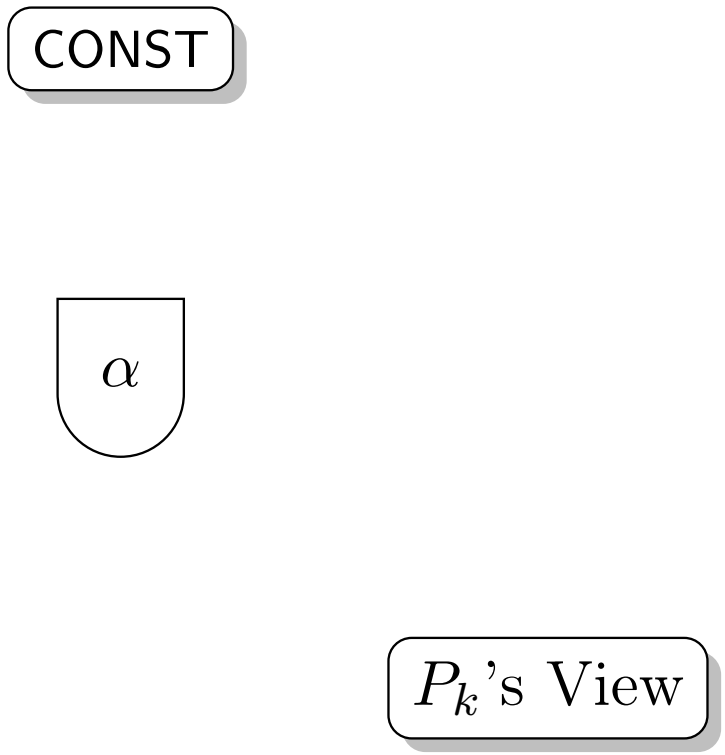
- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.

CONST

P_k's View

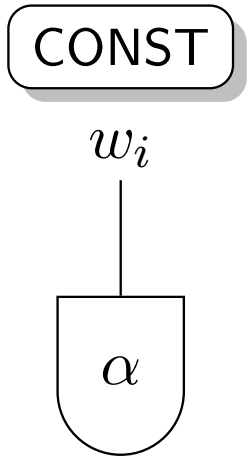
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



BGW Protocol: Evaluation

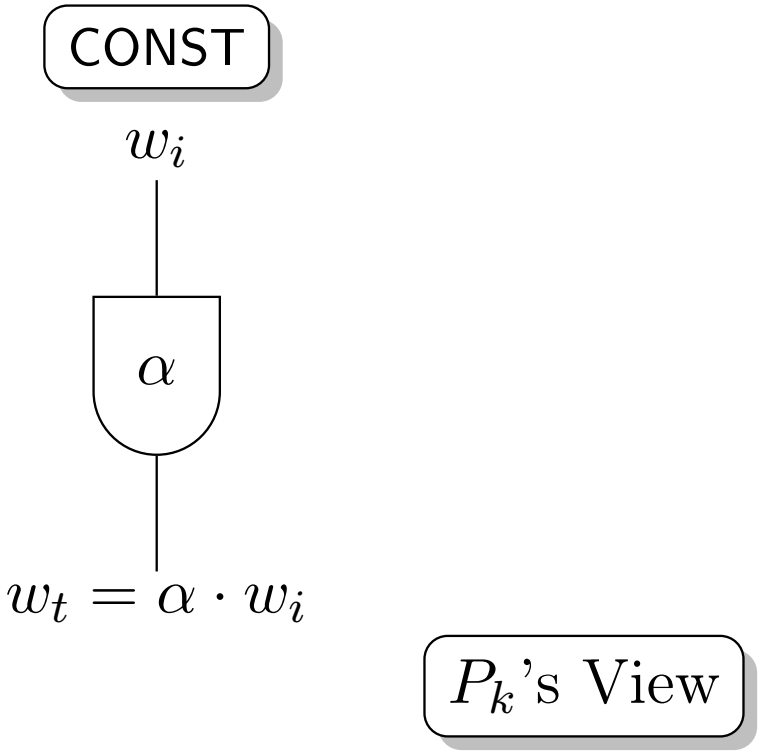
- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



P_k's View

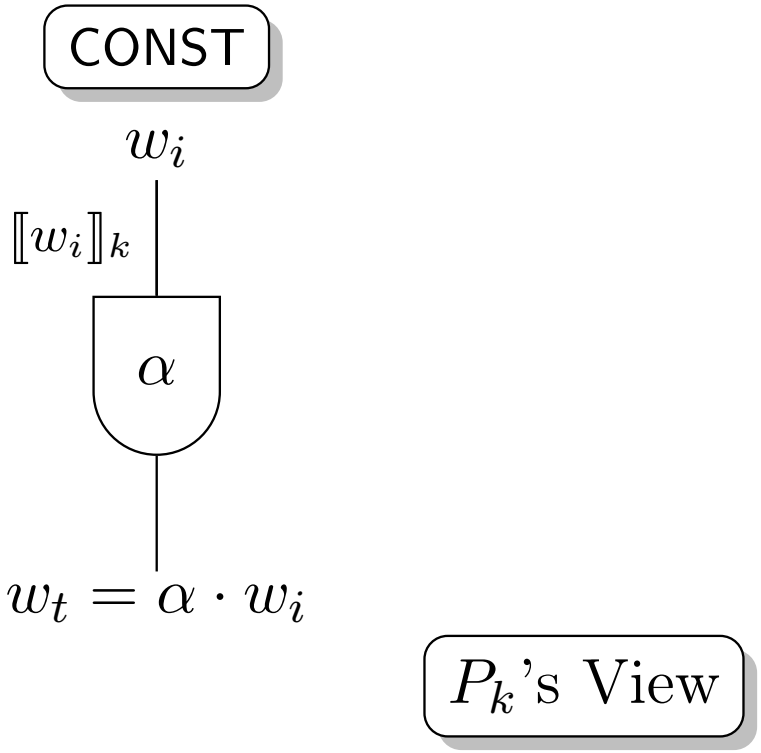
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



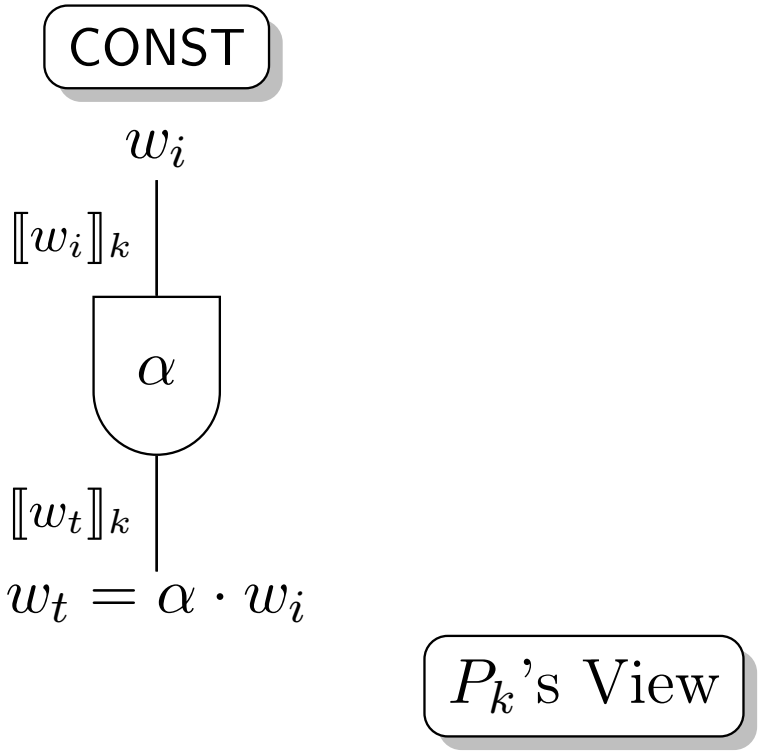
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



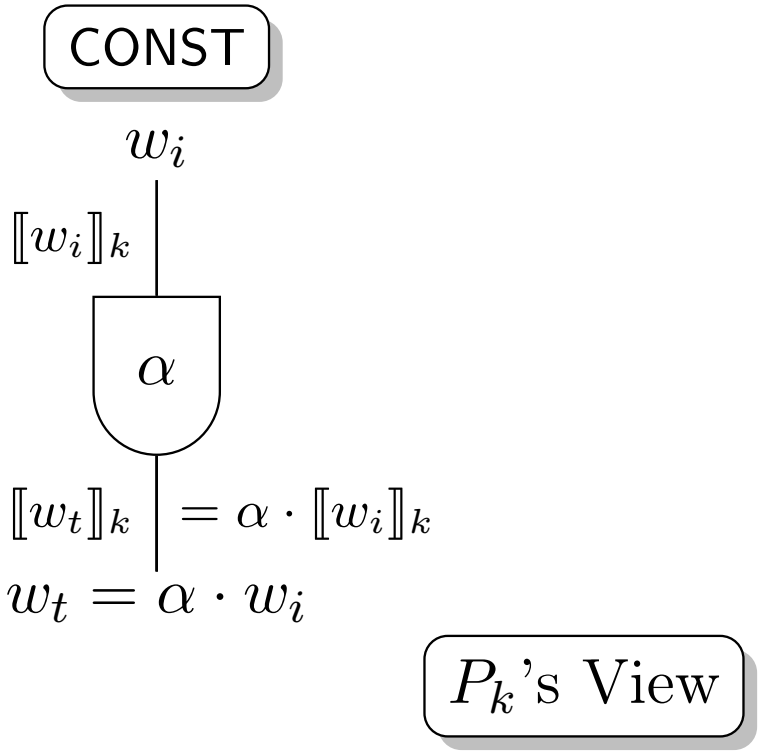
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



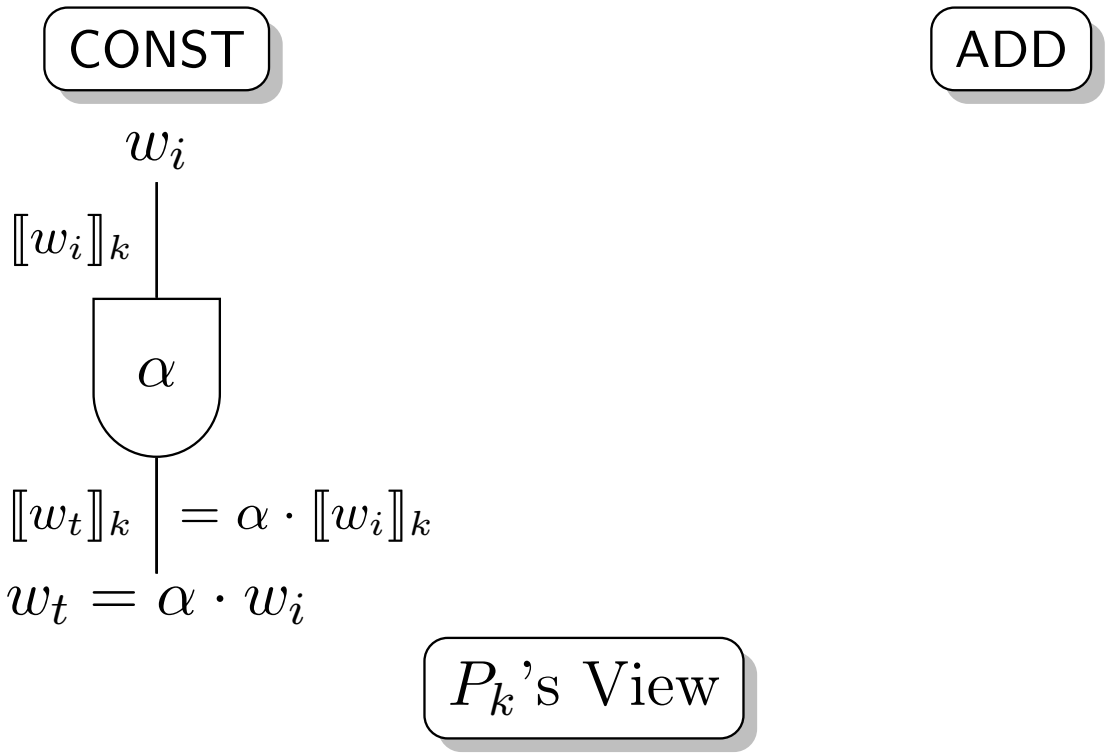
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



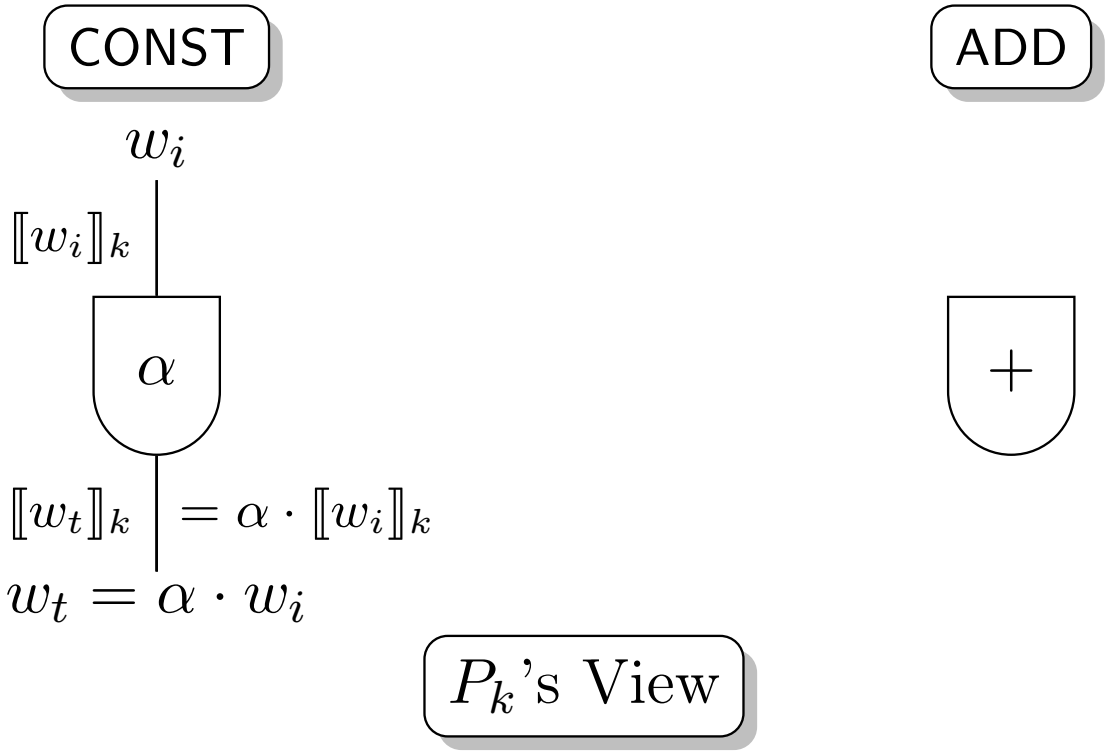
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



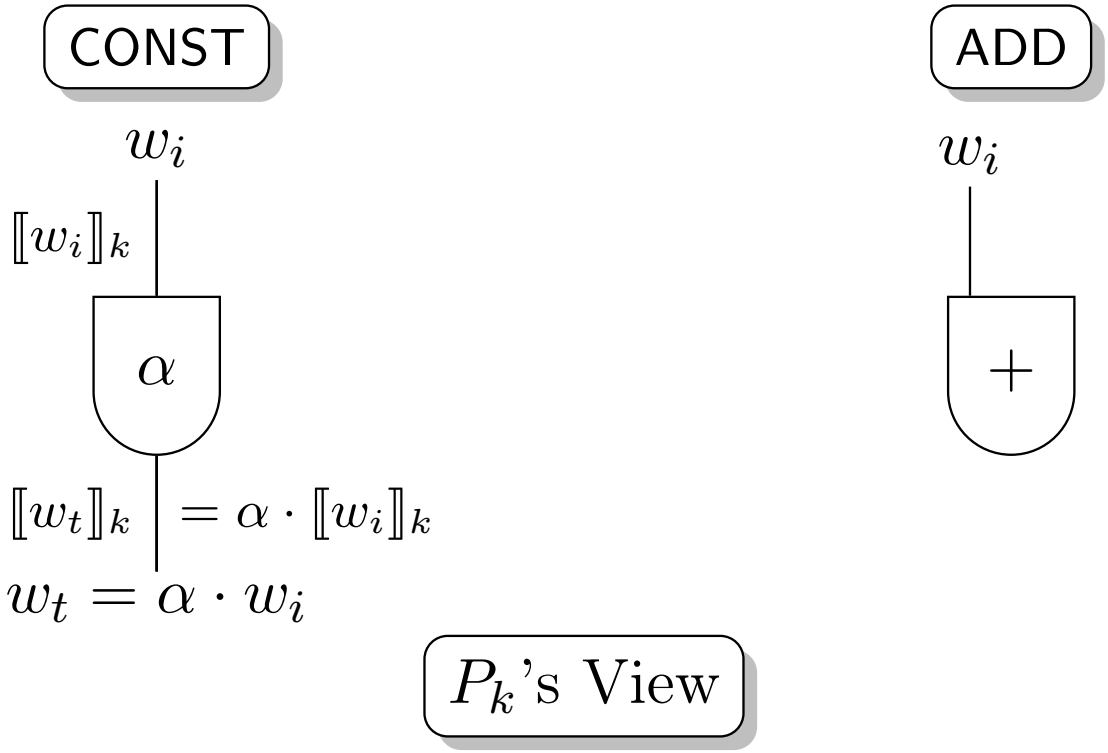
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



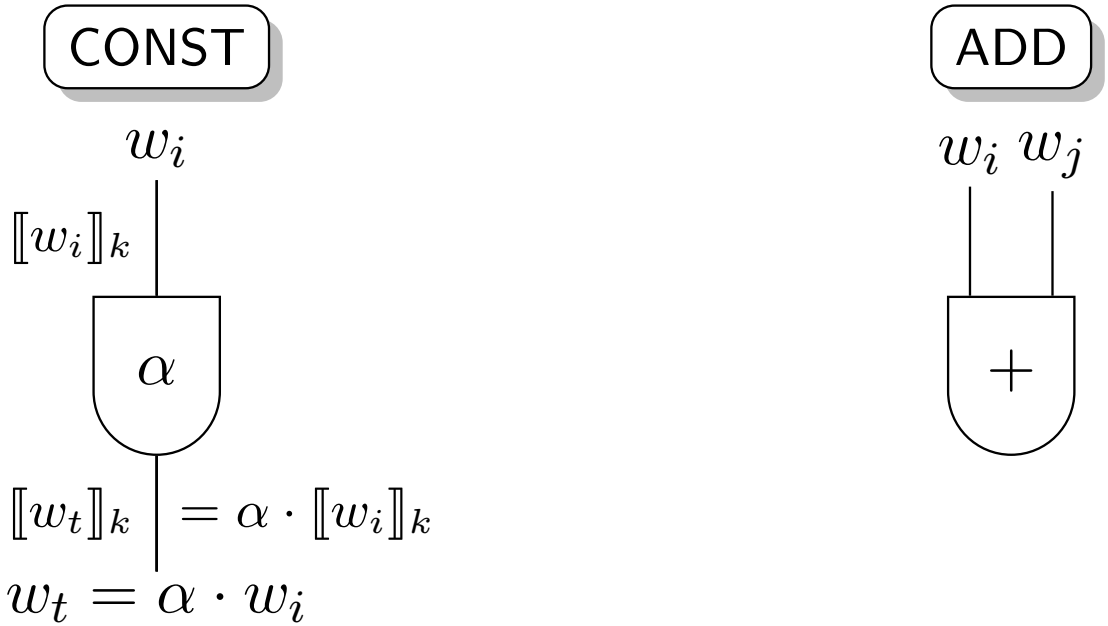
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



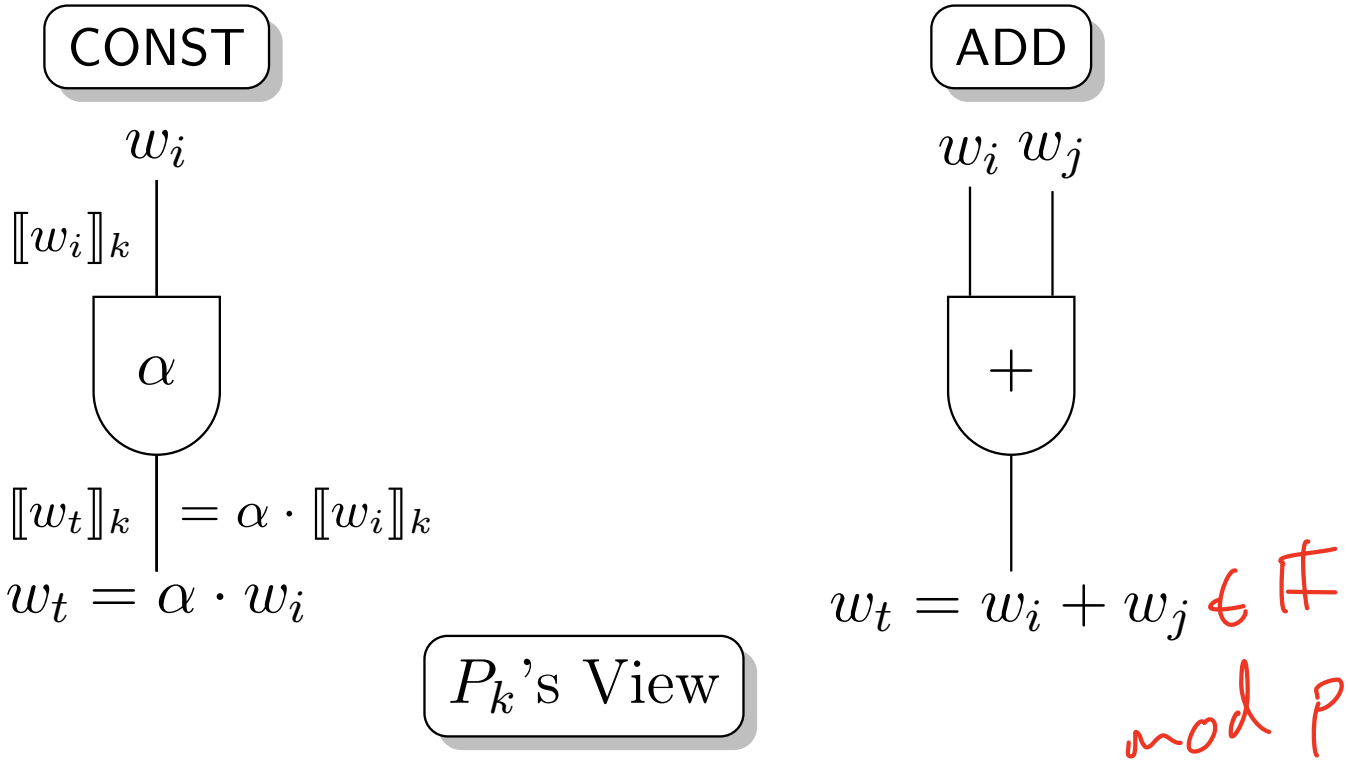
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



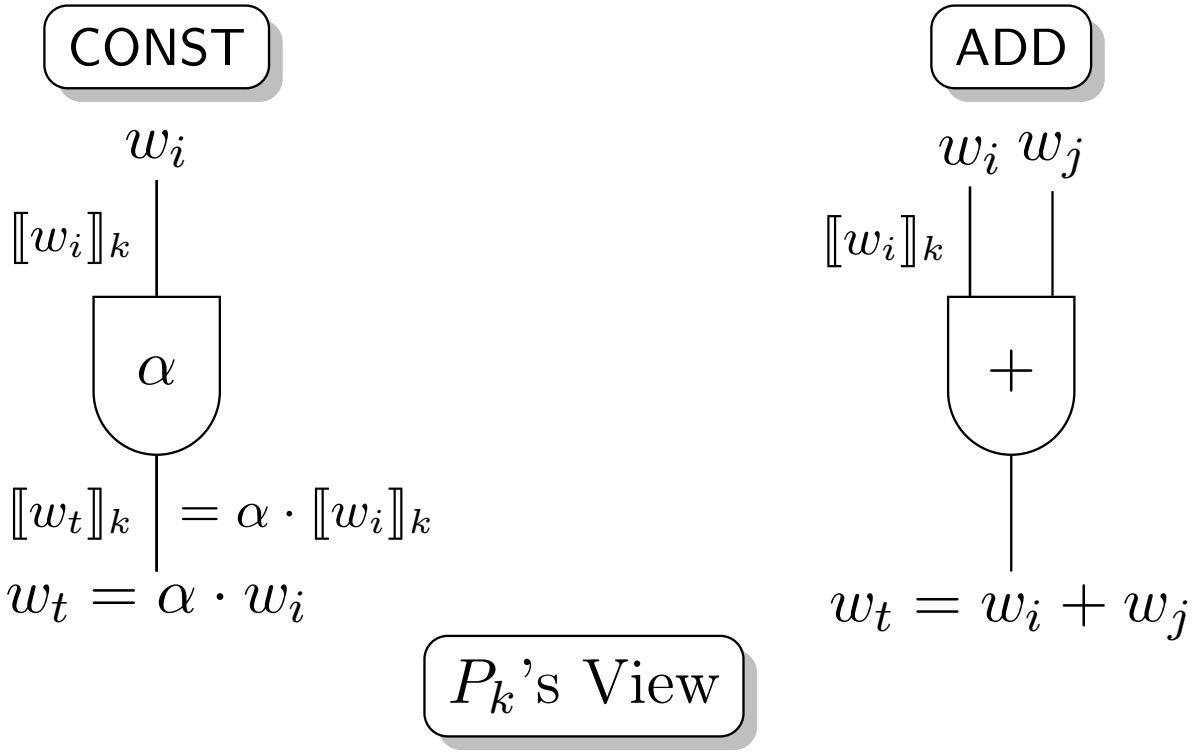
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



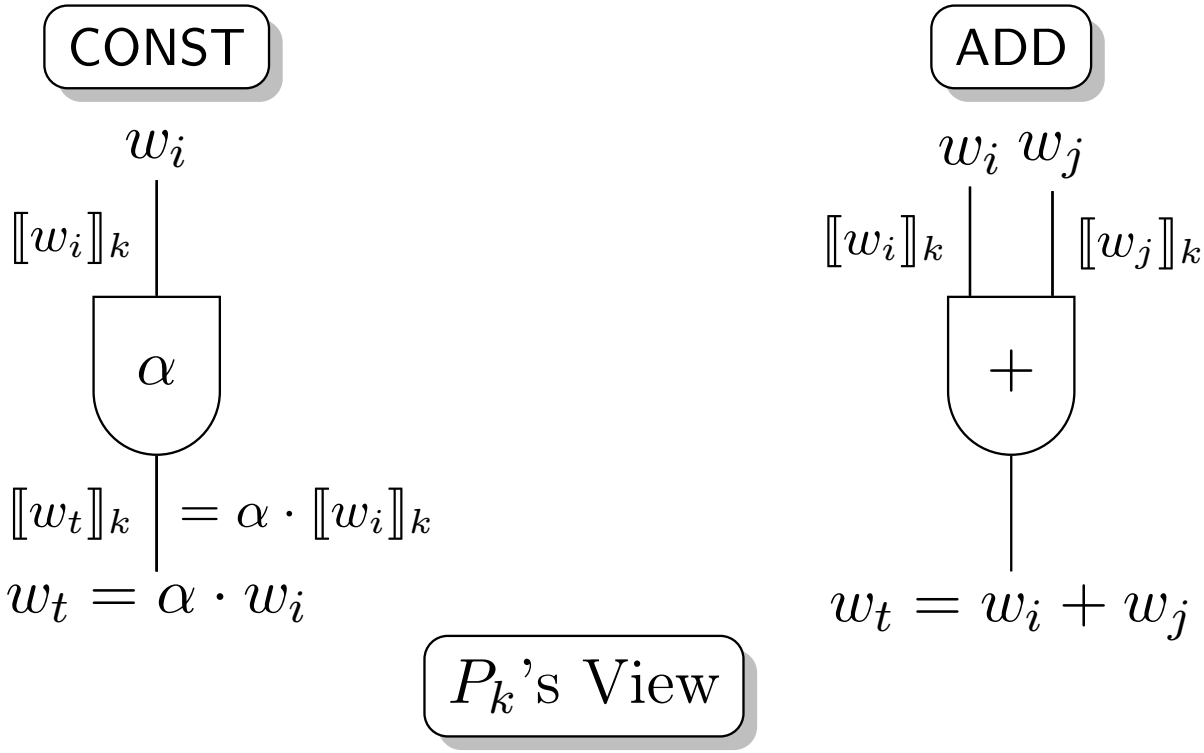
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



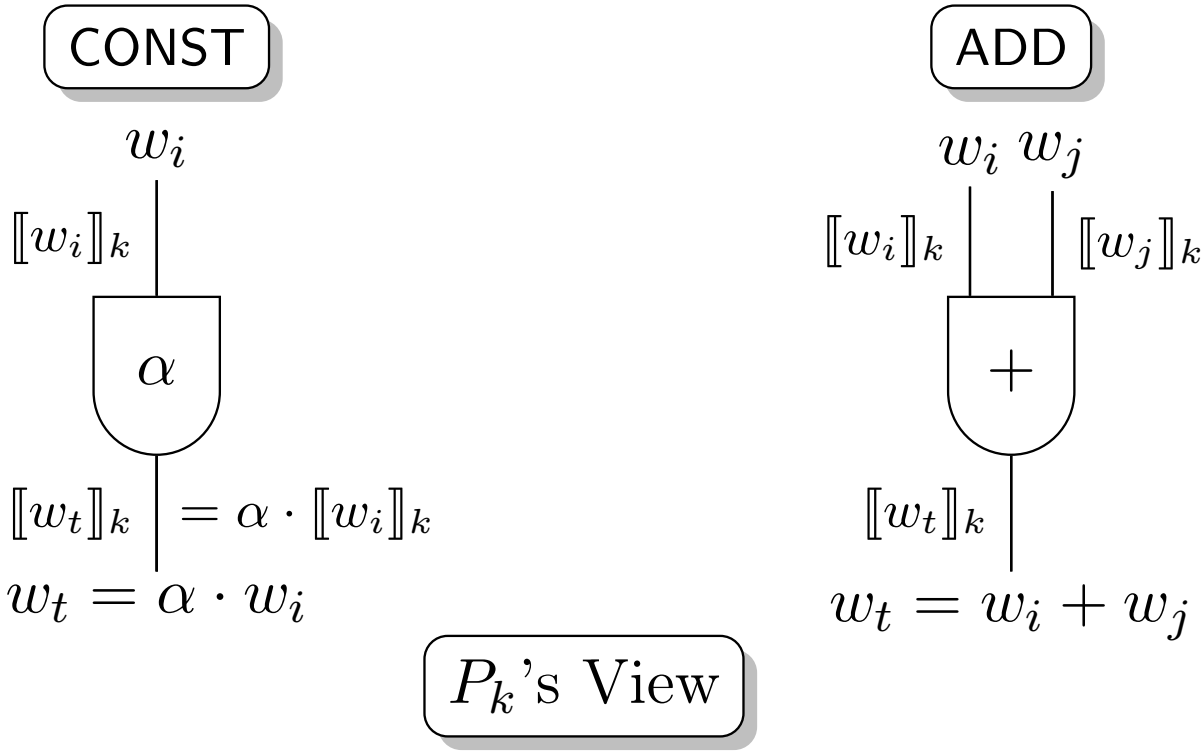
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



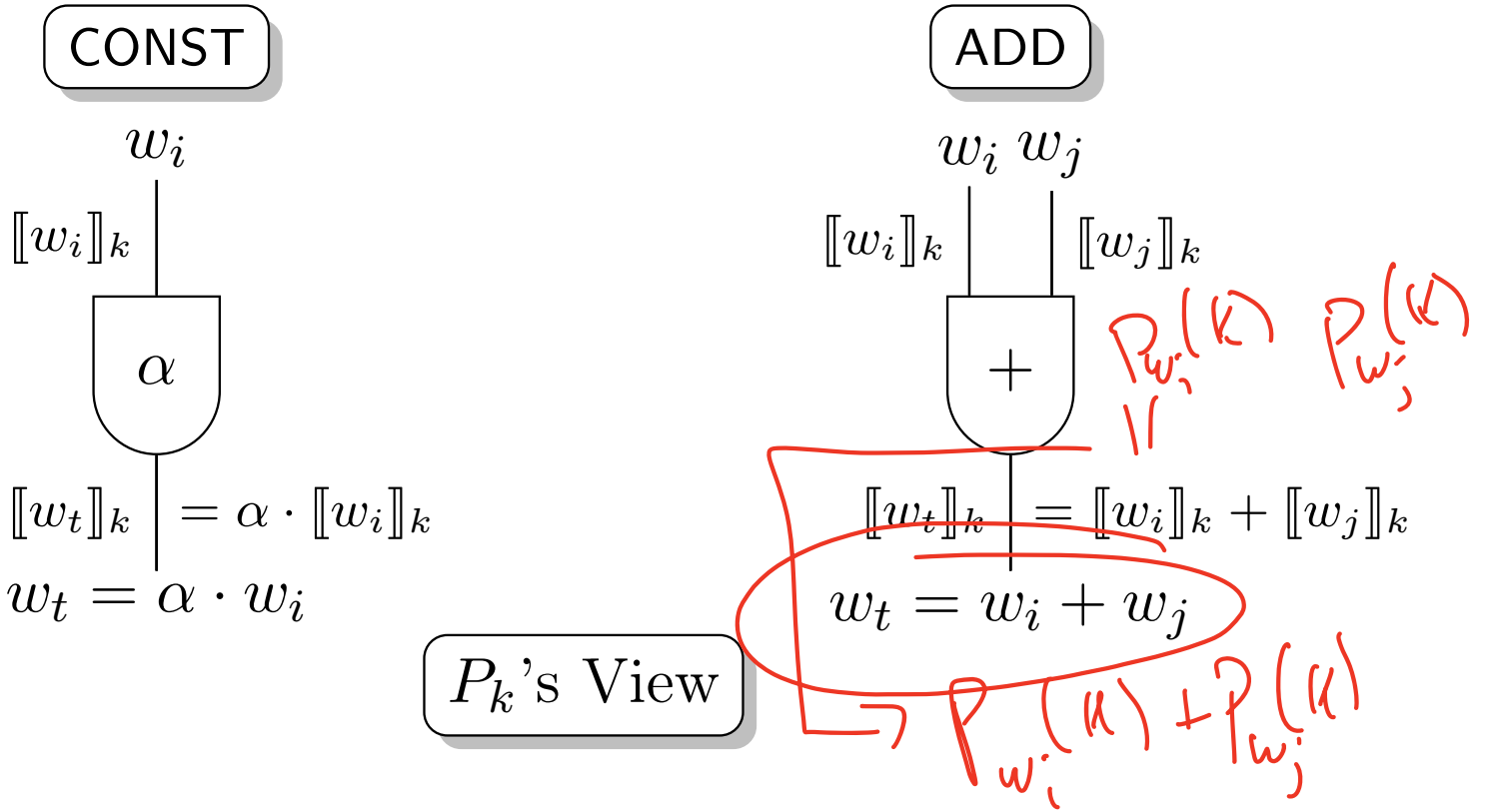
BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



BGW Protocol: Evaluation

- As with GMW, parties in BGW will evaluate the circuit gate-by-gate using the secret shared inputs.
  - E.g., P_i evaluates $\mathcal{C}$ using vector $(\llbracket x_1 \rrbracket_i, \ldots, \llbracket x_m \rrbracket_i)$ as the input.
  - Note this includes $\llbracket x_i \rrbracket_i$.
- As with GMW, ADD gates and CONST gates require no communication among parties.



BGW Protocol: MULT Gates

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

- Parties first compute a *new sharing* $[\![v]\!]_k = [\![w_i]\!]_k \cdot [\![w_j]\!]_k$.

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

- Parties first compute a *new sharing* $[\![v]\!]_k = [\![w_i]\!]_k \cdot [\![w_j]\!]_k$.
  - By properties of Shamir secret sharing, $[\![v]\!]_k = q(k)$, where $q(x) = p_{w_i}(x) \cdot p_{w_j}(x)$.

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

- Parties first compute a *new sharing* $[\![v]\!]_k = [\![w_i]\!]_k \cdot [\![w_j]\!]_k$.
  - By properties of Shamir secret sharing, $[\![v]\!]_k = q(k)$, where $q(x) = p_{w_i}(x) \cdot p_{w_j}(x)$.
  - Notice: $q(0) = p_{w_i}(0) \cdot p_{w_j}(0) = w_i \cdot w_j$, exactly what we want.

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

- Parties first compute a *new sharing* $[\![v]\!]_k = [\![w_i]\!]_k \cdot [\![w_j]\!]_k$.
  - By properties of Shamir secret sharing, $[\![v]\!]_k = q(k)$, where $q(x) = p_{w_i}(x) \cdot p_{w_j}(x)$.
  - Notice: $q(0) = p_{w_i}(0) \cdot p_{w_j}(0) = w_i \cdot w_j$, exactly what we want.
  - Issue: $\deg(q) \leqslant 2t$, which is too large!

BGW Protocol: MULT Gates

- Similar to GMW, MULT gates require interaction.
  - For input wires w_i and w_j, party P_k has $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$
  - Goal: compute $[\![w_t]\!]_k = [\![w_i \cdot w_j]\!]_k$ given the previous secret shares.

- Parties first compute a *new sharing* $[\![v]\!]_k = [\![w_i]\!]_k \cdot [\![w_j]\!]_k$.
  - By properties of Shamir secret sharing, $[\![v]\!]_k = q(k)$, where $q(x) = p_{w_i}(x) \cdot p_{w_j}(x)$.
  - Notice: $q(0) = p_{w_i}(0) \cdot p_{w_j}(0) = w_i \cdot w_j$, exactly what we want.
  - Issue: $\deg(q) \leqslant 2t$, which is too large!

- New Goal: obtain valid secret sharings of $q(0)$ but with threshold t instead of $2t$.

BGW Protocol: Degree Reduction

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

  where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

  where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:
  - Party P_k samples random degree-t polynomial $p_{q(k)}$ such that $p_{q(k)}(0) = q(k)$.

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

  where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:
  - Party P_k samples random degree-t polynomial $p_{q(k)}$ such that $p_{q(k)} = q(k)$.
  - P_k sends $[\![q(k)]\!]_{k'}$ to every party $P_{k'}$ for $k' \neq k$.

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

  where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:
  - Party P_k samples random degree-t polynomial $p_{q(k)}$ such that $p_{q(k)} = q(k)$.
  - P_k sends $[\![q(k)]\!]_{k'}$ to every party $P_{k'}$ for $k' \neq k$.
  - Locally, every P_k now computes $[\![w_t]\!]_k$ as

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:
  - Party P_k samples random degree-t polynomial $p_{q(k)}$ such that $p_{q(k)} = q(k)$.
  - P_k sends $[\![q(k)]\!]_{k'}$ to every party $P_{k'}$ for $k' \neq k$.
  - Locally, every P_k now computes $[\![w_t]\!]_k$ as

$$[\![w_t]\!]_k = \sum_{i=1}^{2t+1} \delta_i(0) \cdot [\![q(i)]\!]_k.$$

BGW Protocol: Degree Reduction

- Main observation: we can write $q(0)$ as follows:

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

where $\delta_i(X)$ is the i^{th} Lagrange Coefficient polynomial.

- Recalling that $[\![v]\!]_k = q(k)$, for every $k \in [m]$:
  - Party P_k samples random degree-t polynomial $p_{q(k)}$ such that $p_{q(k)} = q(k)$.
  - P_k sends $[\![q(k)]\!]_{k'}$ to every party $P_{k'}$ for $k' \neq k$.
  - Locally, every P_k now computes $[\![w_t]\!]_k$ as

$$[\![w_t]\!]_k = \sum_{i=1}^{2t+1} \delta_i(0) \cdot [\![q(i)]\!]_k.$$

- Note, here we require $2t < n$ (i.e., an honest majority) for security.

BGW Protocol: Output Wires and Costs

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties ~~broadcast~~ sends a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties ~~broadcast~~ sends a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.

  - Evaluation: CONST and ADD gates are free.

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties ~~broadcast~~ a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.
  - Evaluation: CONST and ADD gates are free. MULT gates require $O(m^2 \log |\mathbb{F}|)$ communication (same as Setup).

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties broadcast a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.
  - Evaluation: CONST and ADD gates are free. MULT gates require $O(m^2 \log |\mathbb{F}|)$ communication (same as Setup).
  - Output: $O(m^2 \cdot s_{\mathsf{out}} \cdot \log |\mathbb{F}|)$ bits.

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties broadcast a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.
  - Evaluation: CONST and ADD gates are free. MULT gates require $O(m^2 \log |\mathbb{F}|)$ communication (same as Setup).
  - Output: $O(m^2 \cdot s_{\mathsf{out}} \cdot \log |\mathbb{F}|)$ bits.
  - Rounds: requires $2 + \#\mathsf{MULT}(\mathcal{C})$ rounds of communication.

BGW Protocol: Output Wires and Costs

- Output wires: for every output wire w_ℓ
  - Every P_k simply broadcasts $[\![w_\ell]\!]_k$, and all parties reconstruct using Shamir Secret Sharing.

- BGW Costs
  - Setup: All parties broadcast a single field element to every other party, $O(m^2 \log |\mathbb{F}|)$ bits.
  - Evaluation: CONST and ADD gates are free. MULT gates require $O(m^2 \log |\mathbb{F}|)$ communication (same as Setup).
  - Output: $O(m^2 \cdot s_{\mathsf{out}} \cdot \log |\mathbb{F}|)$ bits.
  - Rounds: requires $2 + \#\mathsf{MULT}(\mathcal{C})$ rounds of communication.

- Note: we did not use $\mathcal{F}_{\mathsf{OT}}$!

MPC from Preprocessed Multiplication Triples

Preprocessing MPC

Preprocessing MPC

- Idea: offload expensive online communication to a preprocessing phase.

Preprocessing MPC

- Idea: offload expensive online communication to a preprocessing phase.
  - Preprocessing *must* occur before parties' inputs are known, otherwise security is not guaranteed.

Preprocessing MPC

- Idea: offload expensive online communication to a preprocessing phase.
  - Preprocessing *must* occur before parties' inputs are known, otherwise security is not guaranteed.
- Preprocessing for BGW

Preprocessing MPC

- Idea: offload expensive online communication to a preprocessing phase.
  - Preprocessing *must* occur before parties' inputs are known, otherwise security is not guaranteed.
- Preprocessing for BGW
  - Parties only communicated for MULT gates.

Preprocessing MPC

- Idea: offload expensive online communication to a preprocessing phase.
  - Preprocessing *must* occur before parties' inputs are known, otherwise security is not guaranteed.

- Preprocessing for BGW
  - Parties only communicated for MULT gates.
  - Beaver (1992) showed how to offload most communication to a preprocessing phase.

Beaver Triples

Beaver Triples

- Beaver Triples are simply secret sharings of values a, b, c where $c = a \cdot b$ and $a, b \xleftarrow{\$} \mathbb{F}$.

Beaver Triples

- Beaver Triples are simply secret sharings of values a, b, c where $c = a \cdot b$ and $a, b \xleftarrow{\$} \mathbb{F}$.

- Can be generated in numerous ways

Beaver Triples

- Beaver Triples are simply secret sharings of values a, b, c where $c = a \cdot b$ and $a, b \xleftarrow{\$} \mathbb{F}$.

- Can be generated in numerous ways
  - E.g., running BGW where every party has a random input $r_i \xleftarrow{\$} \mathbb{F}$.

Beaver Triples

- Beaver Triples are simply secret sharings of values a, b, c where $c = a \cdot b$ and $a, b \xleftarrow{\$} \mathbb{F}$.
- Can be generated in numerous ways
  - E.g., running BGW where every party has a random input $r_i \xleftarrow{\$} \mathbb{F}$.
- During the online phase of BGW, we will "consume" one Beaver triple per multiplication gate of $\mathcal{C}$

Preprocessing BGW

Preprocessing BGW

- Preprocessing Phase.

Preprocessing BGW

- Preprocessing Phase.
  - For every multiplication gate G in $\mathcal{C}$, sample $a_G, b_G \xleftarrow{\$} \mathbb{F}$.

Preprocessing BGW

- Preprocessing Phase.
  - For every multiplication gate G in $\mathcal{C}$, sample $a_G, b_G \xleftarrow{\$} \mathbb{F}$.
  - Set $c_G = a_G \cdot b_G$.

Preprocessing BGW

- Preprocessing Phase.
  - For every multiplication gate G in $\mathcal{C}$, sample $a_G, b_G \xleftarrow{\$} \mathbb{F}$.
  - Set $c_G = a_G \cdot b_G$.
  - Using $(t+1, m)$ Shamir secret sharing, send P_k the triple $([\![a_G]\!]_k, [\![b_G]\!]_k, [\![c_G]\!]_k)$.

Preprocessing BGW

- Preprocessing Phase.
  - For every multiplication gate G in $\mathcal{C}$, sample $a_G, b_G \xleftarrow{\$} \mathbb{F}$.
  - Set $c_G = a_G \cdot b_G$.
  - Using $(t+1, m)$ Shamir secret sharing, send P_k the triple $([\![a_G]\!]_k, [\![b_G]\!]_k, [\![c_G]\!]_k)$.

Online Phase

- Parties proceed with the BGW Protocol as normal, except for evaluating MULT gates.

Preprocessing BGW: MULT Gates

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.
- P_k holds $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$.

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.
- P_k holds $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$.
- For every $k \in [m]$:

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.
- P_k holds $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$.
- For every $k \in [m]$:
  - P_k locally computes $[\![w_i - a_G]\!]_k = [\![w_i]\!]_k - [\![a_G]\!]_k$ and $[\![w_j - b_G]\!]_k = [\![w_i]\!]_k - [\![b_G]\!]_k$.

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.
- P_k holds $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$.
- For every $k \in [m]$:
  - P_k locally computes $[\![w_i - a_G]\!]_k = [\![w_i]\!]_k - [\![a_G]\!]_k$ and $[\![w_j - b_G]\!]_k = [\![w_i]\!]_k - [\![a_G]\!]_k$.
  - Each party *broadcasts* their shares $[\![w_i - a_G]\!]_k, [\![w_j - b_G]\!]_k$.

Preprocessing BGW: MULT Gates

- Let G be the current MULT gate with input wires w_i, w_j and output wire w_t.
- P_k holds $[\![w_i]\!]_k$ and $[\![w_j]\!]_k$.
- For every $k \in [m]$:
  - P_k locally computes $[\![w_i - a_G]\!]_k = [\![w_i]\!]_k - [\![a_G]\!]_k$ and $[\![w_j - b_G]\!]_k = [\![w_i]\!]_k - [\![a_G]\!]_k$.
  - Each party *broadcasts* their shares $[\![w_i - a_G]\!]_k, [\![w_j - b_G]\!]_k$.
  - Each party locally reconstructs $d_G = w_i - a_G$ and $e_G = w_j - b_G$.

Interaktion

Preprocessing BGW: MULT Gates

Preprocessing BGW: MULT Gates

- Observe the following identity:

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$w_i \cdot w_j = (w_i - a_G + a_G) \cdot (w_j - b_G + b_G)$$

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \end{aligned}$$

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \end{aligned}$$

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \\ &= d_G e_G + d_G b_G + a_G e_G + c_G. \end{aligned}$$

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} [\![w_t]\!]_k = [\![w_i \cdot w_j]\!] &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \\ &= d_G e_G + d_G b_G + a_G e_G + c_G. \end{aligned}$$

- Thus, P_k locally computes $[\![w_t]\!]_k$ as

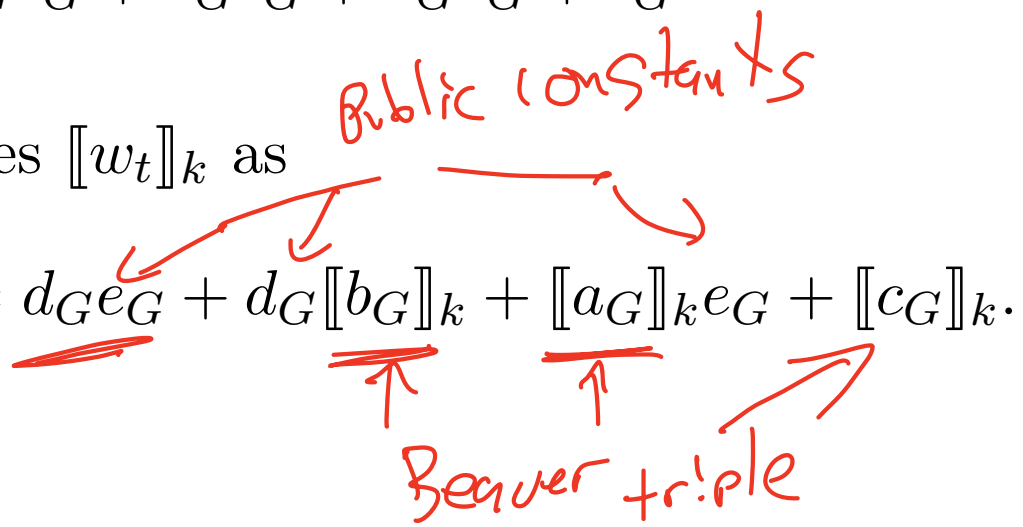
Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \\ &= d_G e_G + d_G b_G + a_G e_G + c_G. \end{aligned}$$

- Thus, P_k locally computes $[\![w_t]\!]_k$ as

$$[\![w_t]\!]_k = [\![w_i \cdot w_j]\!] = d_G e_G + d_G [\![b_G]\!]_k + [\![a_G]\!]_k e_G + [\![c_G]\!]_k.$$



Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \\ &= d_G e_G + d_G b_G + a_G e_G + c_G. \end{aligned}$$

- Thus, P_k locally computes $[\![w_t]\!]_k$ as

$$[\![w_t]\!]_k = [\![w_i \cdot w_j]\!] = d_G e_G + d_G [\![b_G]\!]_k + [\![a_G]\!]_k e_G + [\![c_G]\!]_k.$$

- Improvement over plain BGW: each MULT gate only requires each party to *broadcast* two field elements.

Preprocessing BGW: MULT Gates

- Observe the following identity:

$$\begin{aligned} w_i \cdot w_j &= (w_i - a_G + a_G) \cdot (w_j - b_G + b_G) \\ &= (d_G + a_G) \cdot (e_G + b_G) \\ &= d_G e_G + d_G b_G + a_G e_G + a_G b_G \\ &= d_G e_G + d_G b_G + a_G e_G + c_G. \end{aligned}$$

- Thus, P_k locally computes $[\![w_t]\!]_k$ as

$$[\![w_t]\!]_k = [\![w_i \cdot w_j]\!] = d_G e_G + d_G [\![b_G]\!]_k + [\![a_G]\!]_k e_G + [\![c_G]\!]_k.$$

- Improvement over plain BGW: each MULT gate only requires each party to *broadcast* two field elements.
- Note: using Preprocessing BGW with *additive* secret sharing over $\mathbb{F}$ gives us the (preprocessing) GMW protocol!

Next Time: Indistinguishability Obfuscation

Important Announcements!

- Next week: Spring break.
  - Hope you have a nice break!

- **Friday, April 3, 2026**: Final Project Proposals Due.
  - This counts towards your Final Project Grade, so please don't be late!