UNIT 4

FL with Differential Privacy

FL with Differential Privacy

- Federated Learning (FL) with **Differential Privacy (DP)** is a technique used to enhance the privacy of decentralized machine learning models.
- It ensures that **individual participants' data remains private** while still contributing to the global model.
- This combination is crucial in domains such as healthcare, finance, and smart grid networks, where data privacy is a priority.

Differential Privacy (DP)

- A mathematical framework that adds controlled noise to data or model updates to prevent the identification of individual data points.
- Ensures that an observer cannot determine whether a specific individual's data was used in training.

How FL and DP Work Together

- **Local DP (LDP):** Noise is added at the client level before sending updates to the server.
  - Each user add noise to their data **before** sending it to the server. This provides **stronger privacy** but often results in **lower utility** due to high noise.
- **Central DP (CDP):** The server applies noise to aggregated updates.
  - Data is collected by a trusted server, and noise is added **centrally** (after collection). It usually provides **better utility** but requires trust in the data collector.
- **Hybrid Approaches:** A combination of LDP and CDP for better privacy-utility trade-offs.

Techniques Used

- **Differentially Private Stochastic Gradient Descent (DP-SGD)**: Adds **noise** to gradients during training.
- **Randomized Response**: Clients **randomize their updates** before sending them to the server.
- **Clipping and Noise Addition**: Limits the influence of any single data point and adds Gaussian/Laplace noise.
  - **Why it's used**: Some data points might have **extreme values** that could disproportionately influence the result (e.g., during training).
  - **How it works**: A norm (e.g., **L2 norm**) is applied, and if a data point exceeds a threshold, it's scaled down to the limit.
  - **Example**: In training, if a user's data generates a gradient with **norm 15**, but the **clip norm is set to 5**, the gradient is scaled down proportionally so its norm becomes 5.

Randomize Response

- Instead of directly sharing a sensitive value, a user **applies a probabilistic mechanism** to randomize their response before sending it.
- This makes it difficult for an observer (such as a server in FL) to determine whether a **specific response is the true value** or a randomized one.

Randomize Response

1. **Basic Example (Binary Response)** Suppose a survey asks, *"Have you ever committed tax fraud?"* Instead of answering truthfully, the participant:
    - Flips a coin.
    - If **heads**, they answer truthfully.
    - If **tails**, they flip a second coin:
        - If **heads**, they say *"Yes."*
        - If **tails**, they say *"No."*

    This mechanism ensures plausible deniability while still allowing researchers to estimate the proportion of actual "Yes" responses in a population.

Randomize Response

- **Mathematical Formulation** If a true value $x \in \{0, 1\}$ (e.g., "No" = 0, "Yes" = 1) is reported with probability p and randomized with probability $1 - p$, then:

$$P(\hat{x} = 1) = p \cdot x + (1 - p) \cdot 0.5$$

where $\hat{x}$ is the reported response.

This allows the system to statistically estimate the true distribution while protecting individual privacy.

Where:

- $\hat{x}$ is the reported (randomized) value.
- $x \in \{0, 1\}$ is the true value (e.g., "No" = 0, "Yes" = 1).
- p is the probability of reporting the true value.
- $(1 - p)$ is the probability of randomizing the response.
- The term 0.5 accounts for equal probability of flipping the response when randomization happens.

Randomize Response

Example Calculation

If:

- $p = 0.8$ (80% chance of reporting truthfully),
- $x = 1$ (true answer is Yes),
- $q = 0.5$ (random response is equally likely to be 0 or 1),

Then:

$$P(\hat{x} = 1) = 0.8 \cdot 1 + (1 - 0.8) \cdot 0.5$$

$$P(\hat{x} = 1) = 0.8 + 0.1 = 0.9$$

Thus, there is a **90% probability** that the reported answer will be "Yes" (even though true responses were randomized in some cases).

Randomize Response (Challenges)

- **Accuracy Loss**: Excessive noise can degrade model performance.
- **Parameter Tuning Required**: The probability *p* must be carefully chosen to balance privacy and utility.
- **Aggregation Complexity**: Requires statistical post-processing to reconstruct meaningful insights from noisy data.

Advantages of Randomized Response

✓ **Strong Privacy Protection**: Ensures individual data points remain hidden.

✓ **Simple and Efficient**: Does not require complex cryptographic computations like homomorphic encryption.

✓ **Useful in Decentralized Settings**: Can be applied in federated learning where no trusted third party exists.

Clipping (Gradient Clipping)

Mathematical Formulation

Given a local model update g (a gradient vector) computed by a client, we **clip** it using a pre-defined threshold C:

$$g' = \frac{g}{\max(1, \frac{\|g\|}{C})}$$

Where:

- g is the original gradient,
- C is the clipping norm,
- g' is the clipped gradient.

Why is Clipping Important?

✓ **Prevents Outliers:** Avoids any single client contributing an extremely large gradient, which could reveal sensitive data.
✓ **Improves Training Stability:** Prevents gradient explosion, leading to more stable convergence.
✓ **Enables Differential Privacy:** Clipping ensures that all gradients have a bounded impact before noise is added.

Clipping (Gradient Clipping)

What is Noise Addition?

After clipping, **random noise** is added to the gradients before they are sent to the central server. This ensures **differential privacy (DP)** by making it hard for an adversary to infer information about individual training samples.

Mathematical Formulation

The **noisy gradient** $\tilde{g}$ is obtained by adding noise to the clipped gradient:

$$\tilde{g} = g' + \mathcal{N}(0, \sigma^2 C^2)$$

Where:

- g' is the clipped gradient,
- $\mathcal{N}(0, \sigma^2 C^2)$ is Gaussian noise with **variance** $\sigma^2 C^2$,
- σ controls the noise level (higher σ → stronger privacy, lower accuracy).

Putting It Together: Clipping + Noise Addition in Federated Learning

1. **Clients compute local gradients** based on their private data.
2. **Gradient clipping is applied** to bound the maximum contribution of any client.
3. **Gaussian noise is added** to ensure differential privacy.
4. **Noisy, clipped gradients are sent** to the central server.
5. **The server aggregates gradients** and updates the global model.

Challenges & Trade-offs

🔴 Privacy vs. Utility Trade-off

- More noise (higher σ) → Better privacy, but lower model accuracy.
- Less noise (lower σ) → Higher accuracy, but weaker privacy.

🔴 Choosing an Appropriate Clipping Norm (C)

- **Too high**: Privacy is weaker since gradients are not well-bounded.
- **Too low**: Useful information is lost, degrading model performance.

🔴 Computational Overhead

- Noise addition and clipping introduce additional computations, especially in large-scale federated learning deployments.

Secure Aggregation Techniques

- **Homomorphic Encryption (HE):** Allows computations on encrypted data without decryption, enabling privacy-preserving aggregation in federated learning and cloud environments.
- **Differential Privacy (DP):** Introduces controlled noise to aggregated data to prevent individual data points from being inferred.
- **Secure Multi-Party Computation (SMPC):** Enables multiple parties to jointly compute a function while keeping their inputs private.
- **Secret Sharing:** Data is split into multiple shares distributed among participants, ensuring no single entity can access complete information.
- **Verifiable Computation:** Ensures that aggregated results are correct and tamper-proof using cryptographic proofs.

Encryption Techniques

- **AES (Advanced Encryption Standard):** A widely used symmetric encryption algorithm ensuring data confidentiality.
- **RSA (Rivest-Shamir-Adleman):** An asymmetric encryption technique used for secure key exchange and encryption.
- **Elliptic Curve Cryptography (ECC):** Provides strong security with smaller key sizes, making it efficient for IoT and mobile devices.
- **Fully Homomorphic Encryption (FHE):** Allows complex computations on encrypted data without decryption, ideal for privacy-sensitive applications.
- **Attribute-Based Encryption (ABE):** Encrypts data based on user attributes, providing fine-grained access control.

Homomorphic Encryption

- Homomorphic Encryption (HE) is a form of encryption that allows computations to be **performed on ciphertexts** without decrypting them.
- The results, when decrypted, match those obtained if operations were performed on the plaintext.
- This property is particularly useful in privacy-preserving computations, such as secure cloud computing, encrypted machine learning, and confidential data analysis.

Types of Homomorphic Encryption

Partially Homomorphic Encryption (PHE):

- Supports only a single type of operation (either addition or multiplication).
- Example: RSA and ElGamal encryption support multiplication, while Paillier encryption supports addition.

Somewhat Homomorphic Encryption (SWHE):

- Supports a limited number of both addition and multiplication operations.
- Constraint: The number of operations is restricted due to noise accumulation.

Fully Homomorphic Encryption (FHE):

- Supports an unlimited number of additions and multiplications.
- Enables complex computations on encrypted data without decryption.
- First practical FHE scheme was introduced by Craig Gentry in 2009.

Types of Homomorphic Encryption

Secure Cloud Computing: Users can process encrypted data in the cloud without exposing sensitive information.

Privacy-Preserving Machine Learning: Enables training models on encrypted data without revealing the raw dataset.

Secure Electronic Voting: Ensures confidentiality while allowing mathematical operations to count votes.

Financial and Healthcare Data Security: Protects sensitive records while enabling computations on them.

Blockchain & Secure Transactions: Ensures privacy in smart contracts and decentralized finance (DeFi) applications.

Challenges of Homomorphic Encryption

Computational Overhead: HE operations are significantly slower than plaintext computations.

Key Management: Requires complex key management for security.

Noise Accumulation: Noise grows with each operation, requiring bootstrapping (refreshing ciphertexts).

Key Properties of PHE

- **Single Operation Support:** PHE allows either addition or multiplication but not both.
- **Computational Efficiency:** Compared to FHE, PHE is more efficient since it doesn’t require complex bootstrapping techniques.
- **Security:** PHE schemes are based on hard mathematical problems like integer factorization or discrete logarithms, making them secure under well-known cryptographic assumptions.

PHE

1. Paillier Cryptosystem (Addition Homomorphism)

- Given two encrypted values $Enc(m_1)$ and $Enc(m_2)$, the system allows:

$$Enc(m_1) \cdot Enc(m_2) = Enc(m_1 + m_2)$$

- This means that multiplying encrypted values results in the encryption of their sum.

2. RSA Cryptosystem (Multiplication Homomorphism)

- Given two encrypted values $Enc(m_1)$ and $Enc(m_2)$, the system allows:

$$Enc(m_1) \times Enc(m_2) = Enc(m_1 \cdot m_2)$$

- This means that multiplying ciphertexts results in the encryption of their product.

Applications of PHE

- **Secure Online Voting** – Ensures that votes remain private while allowing addition-based tallying.
- **Privacy-Preserving Finance** – Banks can perform encrypted transactions without accessing user data.
- **Cloud Computing** – Enables cloud providers to process encrypted data without decrypting it.
- **Digital Rights Management (DRM)** – Helps in controlled access and manipulation of encrypted content.

Bootstrapping

- In FHE schemes, every homomorphic operation (addition or multiplication) introduces noise into the ciphertext.
- If too many operations are performed, the noise grows beyond a threshold, making decryption fail.
- **Bootstrapping resets this noise level**, effectively "refreshing" the ciphertext to allow further computations.

How Bootstrapping Works

1. **Encrypt the Encrypted Ciphertext**
    - The noisy ciphertext is re-encrypted using a special procedure.
2. **Evaluate a Decryption Circuit Homomorphically**
    - The system performs a decryption operation on the encrypted ciphertext **while it is still encrypted**.
3. **Output a Fresh, Less Noisy Ciphertext**
    - The resulting ciphertext contains the same message but with reduced noise, allowing continued operations.

Bootstrapping Challenges

- **Computationally Expensive** – Bootstrapping requires executing a complex decryption function homomorphically, which is slow.
- **High Latency** – It takes significant processing power and time, making FHE impractical for real-time applications.
- **Large Key and Ciphertext Size** – Additional complexity leads to larger memory and storage requirements.

Paillier Cryptosystem (Additive Homomorphism)

The **Paillier cryptosystem** supports homomorphic **addition** of encrypted values.

Key Generation:

1. Choose two large prime numbers p and q.
2. Compute $n = p \cdot q$ and $\lambda = \text{lcm}(p-1, q-1)$.
3. Select a generator g such that $g^{\lambda} \mod n^2$ is invertible.
4. Compute the decryption function $\mu = (\text{L}(g^{\lambda} \mod n^2))^{-1} \mod n$, where $\text{L}(x) = \frac{x-1}{n}$.
5. **Public Key:** (n, g), **Private Key:** (λ, μ).

Encryption:

For a message $m \in \mathbb{Z}_n$, select a random number $r \in \mathbb{Z}_n^*$ and compute:

$$Enc(m) = g^m \cdot r^n \mod n^2$$

Decryption:

Given ciphertext c:

$$Dec(c) = \mathrm{L}(c^\lambda \mod n^2) \cdot \mu \mod n$$

Homomorphic Property (Addition):

For two messages m_1, m_2:

$$Enc(m_1) \cdot Enc(m_2) = (g^{m_1} r_1^n) \cdot (g^{m_2} r_2^n) \mod n^2$$

$$= g^{m_1+m_2} (r_1 r_2)^n \mod n^2$$

Since $(r_1 r_2)^n$ is still a random value,

$$Enc(m_1) \cdot Enc(m_2) = Enc(m_1 + m_2)$$

Thus, Paillier encryption **preserves addition** under encryption.

Example

1. **Choose two prime numbers:**

$$p = 7, \quad q = 11$$

2. **Compute n and n^2:**

$$n = p \cdot q = 7 \times 11 = 77$$

$$n^2 = 77^2 = 5929$$

3. **Compute $\lambda(n)$:**

$$\lambda = \text{lcm}(p-1, q-1) = \text{lcm}(6, 10) = 30$$

4. **Choose g:**

   The standard choice is:

$$g = n + 1 = 77 + 1 = 78$$

Example

5. **Compute μ:**

   First, compute:

$$L(g^{\lambda} \mod n^2) = \frac{g^{\lambda} - 1}{n}$$

$$78^{30} \mod 5929 = 4631$$

$$L(4631) = \frac{4631 - 1}{77} = 60$$

   Compute μ as the modular inverse of 60 mod 77:

$$\mu = 60^{-1} \mod 77 = 38$$

Public Key: $(n, g) = (77, 78)$

Private Key: $(\lambda, \mu) = (30, 38)$

Example

Step 2: Encryption of a Message

Let's encrypt $m = 20$.

1. **Choose a random number r:**

   Let $r = 3$.

2. **Compute ciphertext c:**

$$c = g^m \cdot r^n \mod n^2$$

$$c = 78^{20} \cdot 3^{77} \mod 5929$$

Breaking it down:

$$78^{20} \mod 5929 = 2087$$

$$3^{77} \mod 5929 = 1713$$

$$c = (2087 \times 1713) \mod 5929$$

$$c = 3573231 \mod 5929 = 1222$$

So, the **encrypted ciphertext is**:

$$c = 1222$$

Example

Step 3: Homomorphic Addition

Let's encrypt another message $m' = 15$ and compute the sum homomorphically.

1. Encrypt $m' = 15$ with another random $r' = 5$:

$$c' = g^{15} \cdot 5^{77} \mod 5929$$

   Assume:

$$c' = 3210$$

2. Compute **homomorphic addition**:

$$c_{\text{sum}} = c \cdot c' \mod n^2$$

$$c_{\text{sum}} = (1222 \times 3210) \mod 5929$$

$$= 3928620 \mod 5929 = 2513$$

The encrypted sum $Enc(20 + 15) = Enc(35)$ is:

$$c_{\text{sum}} = 2513$$

Example

Step 4: Decryption

Now, let's decrypt $c_{\text{sum}} = 2513$.

1. Compute:

$$L(c_{\text{sum}}^{\lambda} \mod n^2)$$

$$L(2513^{30} \mod 5929)$$

$$2513^{30} \mod 5929 = 2693$$

2. Compute $L(2693)$:

$$L(2693) = \frac{2693 - 1}{77} = 35$$

3. Multiply by μ:

$$m = L(2693) \cdot \mu \mod n$$

$$m = 35 \times 38 \mod 77$$

$$= 1330 \mod 77 = 35$$

Thus, the decrypted message is **35**, which matches $20 + 15$, verifying the homomorphic property.

RSA Cryptosystem (Multiplicative Homomorphism)

The **RSA cryptosystem** supports homomorphic **multiplication** of encrypted values.

Key Generation:

1. Choose two large prime numbers p and q.
2. Compute $n = p \cdot q$ and $\phi(n) = (p-1)(q-1)$.
3. Select a public exponent e such that $\gcd(e, \phi(n)) = 1$.
4. Compute the private exponent d such that $e \cdot d \equiv 1 \mod \phi(n)$.
5. **Public Key:** (n, e), **Private Key:** (d).

RSA Cryptosystem (Multiplicative Homomorphism)

Encryption:

For a message $m \in \mathbb{Z}_n$, compute:

$$Enc(m) = m^e \mod n$$

Decryption:

$$Dec(c) = c^d \mod n = (m^e)^d \mod n = m^{ed} \mod n = m$$

Homomorphic Property (Multiplication):

For two messages m_1, m_2:

$$Enc(m_1) \cdot Enc(m_2) = (m_1^e \mod n) \cdot (m_2^e \mod n) \mod n$$

$$= (m_1 m_2)^e \mod n = Enc(m_1 \cdot m_2)$$

Thus, RSA encryption **preserves multiplication** under encryption.

Secret Sharing

- Secret sharing is a crucial technique used in **secure aggregation** for **federated learning (FL)** to ensure **privacy-preserving model training**.
- Secure aggregation allows multiple clients to collaboratively train a model while keeping their individual model updates private.
- Secret sharing enables a client to split its **model updates (gradients/weights)** into multiple **shares** and distribute them among other participating clients.

Secret Sharing

These shares are structured such that:

- No single client (or a subset of clients below a threshold) can reconstruct the original update.
- Only the **server** (aggregator) can retrieve the sum of all updates without learning individual contributions. (The goal is to allow the **server (aggregator)** to compute the **sum of all local model updates** while ensuring that **individual client contributions remain hidden**.)

How Does This Work?

1. **Clients Secret-Share Their Updates**
    - Each client splits its local update x_i into **multiple shares** using **secret sharing** (e.g., **additive secret sharing** or **Shamir's Secret Sharing**).
    - The shares are distributed among other clients.
2. **Clients Exchange Shares**
    - Each client receives shares from other clients but cannot reconstruct any individual update.
    - The clients securely compute the sum of the shares and send them to the server.
3. **Server Aggregates the Updates**
    - The server only receives the **sum of all shares**, which is equivalent to the sum of the original updates:

$$\sum_{i=1}^{N} x_i$$

    - Since the individual updates are secret-shared, the server cannot learn **any single client's update**.

Why is the Server Unable to Learn Individual Updates?

- **Lack of Access to Individual Shares**:
  The server does not receive individual shares—only the final sum.
- **Randomization**:
  Each client's update is **masked** using randomness, making it impossible to infer individual values.
- **Threshold-Based Security**:
  If a **threshold secret sharing scheme** (e.g., Shamir's Secret Sharing) is used, the server would need at least **t** out of **N** shares to reconstruct any single update—making privacy robust.

Secret Sharing Techniques

Shamir's Secret Sharing (SSS)

- Based on polynomial interpolation.
- A client splits its update into **n shares** using a polynomial of degree **t-1** (where t is the threshold).
- Only **t or more** clients can reconstruct the original update.
- **Advantages:** Strong theoretical security, protects against collusion.
- **Challenges:** High computational overhead, requires exact threshold reconstruction.

Secret Sharing Techniques

SSS is based on **Lagrange interpolation** over a finite field F. The secret is hidden within a **random polynomial**, and only a sufficient number of shares can reconstruct it.

- **Steps in Shamir's Secret Sharing:**

1. **Secret Definition**:
    - Let the secret be a numerical value s (e.g., a model update in FL).
    - Choose a **threshold** t and the **total number of shares** n, where $t \leq n$.

Secret Sharing Techniques

2. **Polynomial Generation**:
    - The client creates a random polynomial of degree $t - 1$:

$$f(x) = s + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$$

      where $a_1, a_2, \ldots, a_{t-1}$ are random coefficients.

3. **Share Distribution**:
    - The client computes n shares:

$$\text{Share}_i = f(x_i) \quad \text{for} \quad i = 1, 2, \ldots, n$$

    - Each participating client C_i receives a unique share $(x_i, f(x_i))$.

Secret Sharing Techniques

4. **Aggregation Phase** (At the Server):
    - The server receives the **sum of all shares** from participating clients.
    - It applies **Lagrange interpolation** to reconstruct the sum of the original model updates:

$$s = f(0) = \sum \lambda_i f(x_i)$$

    - Since the individual shares are masked by the polynomial, the server cannot learn individual updates.

Example of Secret Sharing Techniques

We want to share a secret $S = 42$ among **5 participants**, requiring at least **3 shares** to reconstruct it.

We use a **random polynomial** of degree $t - 1$ (where t is the threshold). Since we require **3 shares to reconstruct the secret**, we choose a **quadratic polynomial** (degree 2):

$$f(x) = a_0 + a_1 x + a_2 x^2$$

where:

- $a_0 = 42$ (the secret)
- a_1 and a_2 are random coefficients (let's pick $a_1 = 7$ and $a_2 = 3$)

Thus, our polynomial is:

$$f(x) = 42 + 7x + 3x^2$$

Example of Secret Sharing Techniques

Step 1: Generate Shares

We compute $f(x)$ for different values of x (choosing $x = 1, 2, 3, 4, 5$):

x	$f(x) = 42 + 7x + 3x^2$	Share $(x, f(x))$
1	$42 + 7(1) + 3(1)^2 = 42 + 7 + 3 = 52$	$(1, 52)$
2	$42 + 7(2) + 3(2)^2 = 42 + 14 + 12 = 68$	$(2, 68)$
3	$42 + 7(3) + 3(3)^2 = 42 + 21 + 27 = 90$	$(3, 90)$
4	$42 + 7(4) + 3(4)^2 = 42 + 28 + 48 = 118$	$(4, 118)$
5	$42 + 7(5) + 3(5)^2 = 42 + 35 + 75 = 152$	$(5, 152)$

So, our **5 shares** are:

$$(1, 52), (2, 68), (3, 90), (4, 118), (5, 152)$$

Example of Secret Sharing Techniques

Step 2: Reconstruct the Secret

We need at least **3 shares** to recover $S = 42$. Let's use $(1, 52), (2, 68), (3, 90)$.

To recover the secret, we use **Lagrange Interpolation**, which reconstructs $f(0)$ (i.e., the constant term $a_0 = 42$).

The Lagrange basis polynomials are:

$$L_i(0) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Using $x_1 = 1, x_2 = 2, x_3 = 3$:

$$S = f(0) = y_1 L_1(0) + y_2 L_2(0) + y_3 L_3(0)$$

Example of Secret Sharing Techniques

Calculating step-by-step:

$$L_1(0) = \frac{(0-2)(0-3)}{(1-2)(1-3)} = \frac{(0-2)(0-3)}{(1-2)(1-3)} = \frac{(0-2)(0-3)}{(1-2)(1-3)} = \frac{6}{2} = 3$$

$$L_2(0) = \frac{(0-1)(0-3)}{(2-1)(2-3)} = \frac{(0-1)(0-3)}{(2-1)(2-3)} = \frac{3}{-1} = -3$$

$$L_3(0) = \frac{(0-1)(0-2)}{(3-1)(3-2)} = \frac{(0-1)(0-2)}{(3-1)(3-2)} = \frac{2}{2} = 1$$

Now, compute S:

$$S = 52(3) + 68(-3) + 90(1)$$

$$= 156 - 204 + 90$$

$$= 42$$

Thus, **we successfully recovered the secret** $S = 42$.

Additive Secret Sharing

Additive Secret Sharing

- Each client splits its update into **random additive shares** and sends them to different clients.
- The sum of all shares reconstructs the original update at the server.
- Used in **Secure Multi-Party Computation (MPC)** protocols like **Secure Aggregation by** Bonawitz et al. (Google FL).
- **Advantages:** Computationally efficient, supports dropout-resilient FL.
- **Challenges:** Requires communication between clients, handling dropout clients increases complexity.

Additive Secret Sharing

Each client C_i **splits its update** x_i into multiple **random shares** so that no single share reveals any information about x_i. Only the **sum of all shares** can reveal the true value.

Steps in Additive Secret Sharing

1. **Secret Splitting (at each client)**
   - Each client C_i has a local update x_i.
   - It generates **random shares** such that:

$$x_i = s_{i,1} + s_{i,2} + \cdots + s_{i,N} \mod P$$

where PPP is a prime number or a large modulus used to prevent overflow.

where:

- $s_{i,1}, s_{i,2}, \ldots, s_{i,N-1}$ are random values.
- $s_{i,N}$ is computed so that the sum equals x_i (modulo a prime P).

Additive Secret Sharing

2. **Share Distribution**
    - Each client C_i **sends shares** to different clients:
        - $s_{i,j}$ is sent to client C_j (for all $j \neq i$).
    - Each client **receives shares** from other clients.
3. **Aggregation (at the Server)**
    - Each client sums up the received shares and sends the result to the server:

$$S_j = \sum_{i=1}^{N} s_{i,j}$$

    - The server **adds up all client sums**:

$$\sum_{j=1}^{N} S_j = \sum_{i=1}^{N} x_i$$

    - Since all intermediate randomness cancels out, the server retrieves the **sum of updates** but not the individual updates.

Additive Secret Sharing

Let's say a model update (e.g., a weight value) is **42**, and we want to split it among three parties.

1. Generate two random numbers:
   - $S_1 = 10$
   - $S_2 = 25$
2. Compute the third share:
   - $S_3 = 42 - (10 + 25) = 7$
3. Distribute shares:
   - Party 1 receives $S_1 = 10$
   - Party 2 receives $S_2 = 25$
   - Party 3 receives $S_3 = 7$
4. Each party processes computations on its share.
5. To reconstruct:
   - $10 + 25 + 7 = 42$

Verifiable Secret Sharing (VSS)

- Verifiable Secret Sharing (VSS) is an enhanced version of **Shamir's Secret Sharing (SSS)** that includes a verification mechanism to prevent malicious participants from providing incorrect shares.

Verifiable Secret Sharing (VSS)

Why is VSS Needed?

While **Shamir's Secret Sharing (SSS)** ensures that a secret is securely divided and only reconstructable with a threshold number of shares, it **does not** verify whether:

✔ The dealer (the entity distributing the shares) is honest.

✔ The shares provided by participants during reconstruction are correct.

Problem Example:

If a malicious party injects incorrect shares into the reconstruction phase, the recovered secret will be wrong.

VSS solves this by adding a verification mechanism that allows participants to **validate their shares** without revealing the secret.

Verifiable Secret Sharing (VSS)

1. **Secret Splitting (Like SSS)**
    - The secret S is encoded in a **random polynomial** $f(x)$.
    - Each participant P_i receives a share (x_i, y_i).
2. **Public Commitments**
    - The dealer publicly commits to the coefficients of the polynomial using a cryptographic commitment scheme (e.g., **Pedersen Commitments** or **Feldman's VSS**).
3. **Verification of Shares**
    - Participants can verify whether their received shares are consistent with the public commitments without learning the secret.
    - If a participant finds an incorrect share, they can reject it before reconstruction.
4. **Secret Reconstruction**
    - Only valid shares participate in reconstructing S, ensuring correctness.

Verifiable Secret Sharing (VSS)

Types of VSS

1. **Feldman's VSS**
    - Uses **public-key cryptography** to commit to polynomial coefficients.
    - Participants verify shares using modular exponentiation.
2. **Pedersen's VSS**
    - Uses **homomorphic commitments** to add an extra layer of secrecy, preventing attackers from inferring secret polynomial coefficients.

Example of Verifiable Secret Sharing (VSS)

Step 3: Compute Public Commitments

We use **modular arithmetic** for verification.

- Choose a **large prime number** $p = 97$.
- Select a **generator** $g = 5$ of a cyclic group.

Compute commitments for each polynomial coefficient:

$$C_0 = g^{a_0} \mod p = 5^{42} \mod 97 = 54$$

$$C_1 = g^{a_1} \mod p = 5^{7} \mod 97 = 78$$

$$C_2 = g^{a_2} \mod p = 5^{3} \mod 97 = 125 \mod 97 = 28$$

Thus, our **public commitments** are:

$$C_0 = 54, \quad C_1 = 78, \quad C_2 = 28$$

Example of Verifiable Secret Sharing (VSS)

Step 4: Verify Shares

Each participant verifies their share (x, y) using the commitment equation:

$$g^{y_i} \stackrel{?}{=} C_0 \cdot C_1^{x_i} \cdot C_2^{x_i^2} \mod p$$

Example Verification for $P_1 = (1, 52)$

$$g^{52} \mod 97 = 5^{52} \mod 97 = 40$$

Now, compute the right-hand side:

$$\begin{aligned} & C_0 \cdot C_1^1 \cdot C_2^{1^2} \mod 97 \\ = & (54 \cdot 78^1 \cdot 28^1) \mod 97 \\ = & (54 \cdot 78 \cdot 28) \mod 97 \\ = & 117936 \mod 97 = 40 \end{aligned}$$

Since both sides are equal:

$$40 = 40$$

P_1's share is **valid** ✅.

Example of Verifiable Secret Sharing (VSS)

Step 5: Secret Reconstruction (Using Lagrange Interpolation)

To reconstruct the secret, we use **any 3 valid shares** and apply **Lagrange Interpolation** (as done in SSS). This will recover:

$$S = 42$$

Thus, **VSS ensures that only valid shares participate in secret reconstruction**.