The **Pairing-Based Cryptography (PBC)** library can be used to implement privacy-preserving mutual authentication schemes by leveraging elliptic curve pairings. PBC is a cryptographic library that enables operations on bilinear pairings, commonly used in identity-based encryption, digital signatures, and authentication protocols.

Below is an example of how you could use PBC to implement a mutual authentication scheme between a **user** and a **server**. This code will show how pairings are used to securely authenticate parties while preserving privacy.

**Generalized Code for User and Server**

#include <pbc/pbc.h>

#include <pbc/pbc\_test.h>

#include <stdio.h>

#include <string.h>

// Key structures for user and server

typedef struct {

element\_t id; // Identity of user

element\_t private\_key; // Private key of user

element\_t public\_key; // Public key of user

} User;

typedef struct {

element\_t secret\_key; // Server's secret key

element\_t public\_key; // Server's public key

} Server;

// Setup function (initializing pairing and server keys)

void setup(pairing\_t pairing, Server \*server) {

// Generate a random element for server's secret key

element\_init\_Zr(server->secret\_key, pairing);

element\_random(server->secret\_key);

// Generate server's public key as a pairing of the secret key

element\_init\_G1(server->public\_key, pairing);

element\_pow\_zn(server->public\_key, server->secret\_key, pairing);

}

// User registration (User generates keys)

void user\_registration(pairing\_t pairing, User \*user, Server \*server) {

// Set user's identity as a random element

element\_init\_G1(user->id, pairing);

element\_random(user->id);

// Generate user's private key as secret key \* identity

element\_init\_G1(user->private\_key, pairing);

element\_pow\_zn(user->private\_key, user->id, server->secret\_key);

// Public key is identity

element\_init\_G1(user->public\_key, pairing);

element\_set(user->public\_key, user->id);

}

// Mutual authentication between user and server

int authenticate\_user(pairing\_t pairing, User \*user, Server \*server) {

element\_t server\_computation;

element\_t user\_computation;

// Server computes e(PK\_user, SK\_server)

element\_init\_GT(server\_computation, pairing);

pairing\_apply(server\_computation, user->public\_key, server->secret\_key, pairing);

// User computes e(ID\_user, SK\_server)

element\_init\_GT(user\_computation, pairing);

pairing\_apply(user\_computation, user->private\_key, server->public\_key, pairing);

// Compare both computations

if (element\_cmp(server\_computation, user\_computation) == 0) {

printf("Mutual authentication succeeded.\n");

return 1;

} else {

printf("Authentication failed.\n");

return 0;

}

}

int main() {

// Initialize pairing (Type A pairing as example)

pairing\_t pairing;

pbc\_demo\_pairing\_init(pairing, NULL);

// Server and user structures

Server server;

User user;

// Setup phase (server generates secret and public keys)

setup(pairing, &server);

// User registration (user generates private key based on server's secret key)

user\_registration(pairing, &user, &server);

// Perform mutual authentication

authenticate\_user(pairing, &user, &server);

// Clear elements and pairing

element\_clear(user.private\_key);

element\_clear(user.public\_key);

element\_clear(server.secret\_key);

element\_clear(server.public\_key);

pairing\_clear(pairing);

return 0;

}

**Explanation of the Code:**

1. **Pairing Setup**: The setup() function initializes the pairing parameters and generates the server's secret and public keys. The server's public key is derived from the pairing of the secret key and an elliptic curve group.
2. **User Registration**: In user\_registration(), the user generates a private key based on the server's secret key and the user's identity. The private key is a pairing of the server's secret key and the user's identity.
3. **Authentication Process**:
   * The server computes a pairing of the user's public key (identity) with the server's secret key.
   * The user computes a pairing of their private key with the server's public key.
   * If the computed pairings match, mutual authentication is successful.
4. **Element Operations**:
   * element\_init\_\*(): Initializes elements for operations.
   * element\_random(): Generates random values for identity and secret key.
   * pairing\_apply(): Computes the bilinear pairing between two group elements.

**Compilation:**

To compile the above code, use:

gcc -o mutual\_auth mutual\_auth.c -lpbc -lgmp

**Key Components:**

* **Bilinear Pairings**: The authentication relies on the bilinear property of the pairing function, ensuring that computations between the user’s private key and the server’s public key match the server’s computation.
* **Security**: The scheme leverages the difficulty of solving bilinear Diffie-Hellman problems to ensure secure and privacy-preserving mutual authentication.

**CODE For All the entities in the system using PBC**

To implement ALMASH using the **PBC (Pairing-Based Cryptography)** library, we simulated the authentication process involving four entities: **Server**, **User**, **Things (IoT healthcare devices)**, and a **Registration** **Gateway**.

**Entities:**

1. **Server**: Verifies user authentication and provides services.
2. **User**: Requests services from the server via a gateway.
3. **Things (IoT healthcare Devices)**: Medical devices connected through the gateway.
4. **Registration Gateway**: Acts as a middle layer between users, IoT devices, and the server.

**Scheme Overview:**

1. **User Authentication**: The user is authenticated anonymously to the server through the gateway.
2. **Things Registration/Authentication**: IoT devices authenticate to the server and verify user permissions via the gateway.
3. **Anonymity**: The scheme uses pairings to ensure anonymity by hiding the identity of users and things.
4. **Lightweight Operation**: Uses elliptic curve operations and pairings to minimize computation overhead

.

#include <pbc/pbc.h>

#include <pbc/pbc\_test.h>

#include <stdio.h>

#include <string.h>

#include <gmp.h> // GNU MP library for big number arithmetic

// Define entity structures

typedef struct {

element\_t id; // Identity of the user

element\_t private\_key; // User's private key

element\_t public\_key; // User's public key (anonymized)

} User;

typedef struct {

element\_t id; // Identity of IoT device (Thing)

element\_t private\_key; // Thing's private key

element\_t public\_key; // Thing's public key

} Thing;

typedef struct {

element\_t secret\_key; // Server's secret key

element\_t public\_key; // Server's public key

element\_t shared\_secret; // Shared secret distributed via Shamir's Secret Sharing

} Server;

typedef struct {

element\_t shared\_key; // Gateway's shared key (computed from Shamir's Secret Sharing)

element\_t partial\_key; // Gateway's part of Shamir's Secret

} Gateway;

// Shamir Secret Sharing: Splitting the server's secret into parts

void shamir\_secret\_sharing(pairing\_t pairing, Server \*server, Gateway \*gateway, int t, int n) {

// Shamir Secret Sharing Parameters: (t, n) threshold scheme

// Generate `n` shares of the secret and distribute them to gateways

// Generate the secret (Server's private key is the secret)

element\_init\_Zr(server->shared\_secret, pairing);

element\_random(server->shared\_secret);

// Simulate distribution of shares (simplified for this implementation)

// In practice, shares would be distributed to `n` entities.

element\_init\_Zr(gateway->partial\_key, pairing); // Store the gateway's part

element\_set(gateway->partial\_key, server->shared\_secret); // For simplicity, use the whole secret for now

}

// Setup function: Initializes system and keys for the server

void setup(pairing\_t pairing, Server \*server) {

// Generate server's secret key and public key

element\_init\_Zr(server->secret\_key, pairing);

element\_random(server->secret\_key);

element\_init\_G1(server->public\_key, pairing);

element\_pow\_zn(server->public\_key, server->public\_key, server->secret\_key);

}

// User registration: User generates private and public keys using ECC

void user\_registration(pairing\_t pairing, User \*user, Server \*server) {

element\_init\_G1(user->id, pairing);

element\_random(user->id); // User identity is anonymized (random)

// Generate user's private key based on server's secret key

element\_init\_G1(user->private\_key, pairing);

element\_pow\_zn(user->private\_key, user->id, server->secret\_key);

// Public key (anonymized version)

element\_init\_G1(user->public\_key, pairing);

element\_set(user->public\_key, user->id); // Anonymized

}

// Thing (IoT device) registration: Generates keys based on ECC and server's secret key

void thing\_registration(pairing\_t pairing, Thing \*thing, Server \*server) {

element\_init\_G1(thing->id, pairing);

element\_random(thing->id); // Thing's identity is anonymized

// Generate Thing's private key based on server's secret key

element\_init\_G1(thing->private\_key, pairing);

element\_pow\_zn(thing->private\_key, thing->id, server->secret\_key);

// Public key of Thing

element\_init\_G1(thing->public\_key, pairing);

element\_set(thing->public\_key, thing->id); // Anonymized

}

// Gateway registration: Gateway shares a key with server via Shamir Secret Sharing

void gateway\_registration(pairing\_t pairing, Gateway \*gateway, Server \*server) {

// Shared key derived from Shamir's secret (in a real system, t-out-of-n shares would be used)

element\_init\_G1(gateway->shared\_key, pairing);

element\_pow\_zn(gateway->shared\_key, server->public\_key, server->secret\_key);

}

// User authentication via gateway to server (using ECC and Shamir)

int authenticate\_user\_via\_gateway(pairing\_t pairing, User \*user, Gateway \*gateway, Server \*server) {

element\_t server\_computation;

element\_t user\_computation;

// Server computes pairing of user's public key and server's secret key

element\_init\_GT(server\_computation, pairing);

pairing\_apply(server\_computation, user->public\_key, server->secret\_key, pairing);

// User computes shared key with gateway (using ECC and shared secret)

element\_init\_GT(user\_computation, pairing);

pairing\_apply(user\_computation, user->private\_key, gateway->shared\_key, pairing);

// If the two computed values match, authentication succeeds

if (element\_cmp(server\_computation, user\_computation) == 0) {

printf("User successfully authenticated via gateway.\n");

return 1;

} else {

printf("User authentication failed.\n");

return 0;

}

}

// Thing authentication to the server via gateway (using ECC and Shamir)

int authenticate\_thing\_via\_gateway(pairing\_t pairing, Thing \*thing, Gateway \*gateway, Server \*server) {

element\_t server\_computation;

element\_t thing\_computation;

// Server computes pairing of Thing's public key and server's secret key

element\_init\_GT(server\_computation, pairing);

pairing\_apply(server\_computation, thing->public\_key, server->secret\_key, pairing);

// Thing computes shared key with gateway (using ECC and shared secret)

element\_init\_GT(thing\_computation, pairing);

pairing\_apply(thing\_computation, thing->private\_key, gateway->shared\_key, pairing);

// If the two computed values match, authentication succeeds

if (element\_cmp(server\_computation, thing\_computation) == 0) {

printf("Thing successfully authenticated via gateway.\n");

return 1;

} else {

printf("Thing authentication failed.\n");

return 0;

}

}

// Main function

int main() {

// Initialize pairing

pairing\_t pairing;

pbc\_demo\_pairing\_init(pairing, NULL);

// Create server, user, thing, and gateway structures

Server server;

User user;

Thing thing;

Gateway gateway;

// Setup phase: Server generates public and private keys

setup(pairing, &server);

// Shamir Secret Sharing: Distribute shared secrets

shamir\_secret\_sharing(pairing, &server, &gateway, 2, 3); // Threshold = 2, Number of shares = 3

// Register user and thing with the server

user\_registration(pairing, &user, &server);

thing\_registration(pairing, &thing, &server);

// Gateway shares a key with the server

gateway\_registration(pairing, &gateway, &server);

// Authenticate user via the gateway

authenticate\_user\_via\_gateway(pairing, &user, &gateway, &server);

// Authenticate Thing (IoT device) via the gateway

authenticate\_thing\_via\_gateway(pairing, &thing, &gateway, &server);

// Clear elements and pairing

element\_clear(user.private\_key);

element\_clear(user.public\_key);

element\_clear(thing.private\_key);

element\_clear(thing.public\_key);

element\_clear(server.secret\_key);

element\_clear(server.public\_key);

element\_clear(gateway.shared\_key);

pairing\_clear(pairing);

return 0;

}

In mutual authentication, both sides (e.g., user to gateway, gateway to user, user to server) verify each other to ensure the integrity and authenticity of the communication. In this scheme:

1. User authenticates to the Gateway and Server.
2. Gateway authenticates to the User and Server.
3. Things (IoT devices) authenticate to the Gateway and Server.
4. Server authenticates to all other entities (User, Gateway, and Things).

We'll extend the PBC code to incorporate mutual authentication. Here's the enhanced version with the mutual authentication logic:

#include <pbc/pbc.h>

#include <pbc/pbc\_test.h>

#include <stdio.h>

#include <string.h>

#include <gmp.h> // GNU MP library for big number arithmetic

// Define entity structures

typedef struct {

element\_t id; // Identity of the user

element\_t private\_key; // User's private key

element\_t public\_key; // User's public key (anonymized)

} User;

typedef struct {

element\_t id; // Identity of IoT device (Thing)

element\_t private\_key; // Thing's private key

element\_t public\_key; // Thing's public key

} Thing;

typedef struct {

element\_t secret\_key; // Server's secret key

element\_t public\_key; // Server's public key

element\_t shared\_secret; // Shared secret distributed via Shamir's Secret Sharing

} Server;

typedef struct {

element\_t shared\_key; // Gateway's shared key (computed from Shamir's Secret Sharing)

element\_t partial\_key; // Gateway's part of Shamir's Secret

element\_t public\_key; // Gateway's public key

element\_t private\_key; // Gateway's private key

} Gateway;

// Shamir Secret Sharing: Splitting the server's secret into parts

void shamir\_secret\_sharing(pairing\_t pairing, Server \*server, Gateway \*gateway, int t, int n) {

// Generate the secret (Server's private key is the secret)

element\_init\_Zr(server->shared\_secret, pairing);

element\_random(server->shared\_secret);

// Simulate distribution of shares (simplified for this implementation)

// In practice, shares would be distributed to `n` entities.

element\_init\_Zr(gateway->partial\_key, pairing); // Store the gateway's part

element\_set(gateway->partial\_key, server->shared\_secret); // For simplicity, use the whole secret for now

}

// Setup function: Initializes system and keys for the server

void setup(pairing\_t pairing, Server \*server) {

// Generate server's secret key and public key

element\_init\_Zr(server->secret\_key, pairing);

element\_random(server->secret\_key);

element\_init\_G1(server->public\_key, pairing);

element\_pow\_zn(server->public\_key, server->public\_key, server->secret\_key);

}

// User registration: User generates private and public keys using ECC

void user\_registration(pairing\_t pairing, User \*user, Server \*server) {

element\_init\_G1(user->id, pairing);

element\_random(user->id); // User identity is anonymized (random)

// Generate user's private key based on server's secret key

element\_init\_G1(user->private\_key, pairing);

element\_pow\_zn(user->private\_key, user->id, server->secret\_key);

// Public key (anonymized version)

element\_init\_G1(user->public\_key, pairing);

element\_set(user->public\_key, user->id); // Anonymized

}

// Thing (IoT device) registration: Generates keys based on ECC and server's secret key

void thing\_registration(pairing\_t pairing, Thing \*thing, Server \*server) {

element\_init\_G1(thing->id, pairing);

element\_random(thing->id); // Thing's identity is anonymized

// Generate Thing's private key based on server's secret key

element\_init\_G1(thing->private\_key, pairing);

element\_pow\_zn(thing->private\_key, thing->id, server->secret\_key);

// Public key of Thing

element\_init\_G1(thing->public\_key, pairing);

element\_set(thing->public\_key, thing->id); // Anonymized

}

// Gateway registration: Gateway shares a key with server via Shamir Secret Sharing

void gateway\_registration(pairing\_t pairing, Gateway \*gateway, Server \*server) {

// Generate Gateway's private and public key

element\_init\_G1(gateway->private\_key, pairing);

element\_random(gateway->private\_key);

element\_init\_G1(gateway->public\_key, pairing);

element\_pow\_zn(gateway->public\_key, gateway->private\_key, server->secret\_key);

// Shared key derived from Shamir's secret (in a real system, t-out-of-n shares would be used)

element\_init\_G1(gateway->shared\_key, pairing);

element\_pow\_zn(gateway->shared\_key, server->public\_key, server->secret\_key);

}

// Mutual authentication: Gateway and User authenticate each other

int mutual\_authentication\_user\_gateway(pairing\_t pairing, User \*user, Gateway \*gateway) {

element\_t auth\_value1, auth\_value2;

// Gateway computes pairing of user's public key and gateway's private key

element\_init\_GT(auth\_value1, pairing);

pairing\_apply(auth\_value1, user->public\_key, gateway->private\_key, pairing);

// User computes pairing of gateway's public key and user's private key

element\_init\_GT(auth\_value2, pairing);

pairing\_apply(auth\_value2, gateway->public\_key, user->private\_key, pairing);

// If both values match, mutual authentication succeeds

if (element\_cmp(auth\_value1, auth\_value2) == 0) {

printf("User and Gateway successfully authenticated each other.\n");

return 1;

} else {

printf("User-Gateway mutual authentication failed.\n");

return 0;

}

}

// Mutual authentication between Thing and Gateway

int mutual\_authentication\_thing\_gateway(pairing\_t pairing, Thing \*thing, Gateway \*gateway) {

element\_t auth\_value1, auth\_value2;

// Gateway computes pairing of thing's public key and gateway's private key

element\_init\_GT(auth\_value1, pairing);

pairing\_apply(auth\_value1, thing->public\_key, gateway->private\_key, pairing);

// Thing computes pairing of gateway's public key and thing's private key

element\_init\_GT(auth\_value2, pairing);

pairing\_apply(auth\_value2, gateway->public\_key, thing->private\_key, pairing);

// If both values match, mutual authentication succeeds

if (element\_cmp(auth\_value1, auth\_value2) == 0) {

printf("Thing and Gateway successfully authenticated each other.\n");

return 1;

} else {

printf("Thing-Gateway mutual authentication failed.\n");

return 0;

}

}

// Mutual authentication between Server and Gateway

int mutual\_authentication\_server\_gateway(pairing\_t pairing, Server \*server, Gateway \*gateway) {

element\_t auth\_value1, auth\_value2;

// Server computes pairing of gateway's public key and server's secret key

element\_init\_GT(auth\_value1, pairing);

pairing\_apply(auth\_value1, gateway->public\_key, server->secret\_key, pairing);

// Gateway computes pairing of server's public key and gateway's private key

element\_init\_GT(auth\_value2, pairing);

pairing\_apply(auth\_value2, server->public\_key, gateway->private\_key, pairing);

// If both values match, mutual authentication succeeds

if (element\_cmp(auth\_value1, auth\_value2) == 0) {

printf("Server and Gateway successfully authenticated each other.\n");

return 1;

} else {

printf("Server-Gateway mutual authentication failed.\n");

return 0;

}

}

// Main function

int main() {

// Initialize pairing

pairing\_t pairing;

pbc\_demo\_pairing\_init(pairing, NULL);

// Create server, user, thing, and gateway structures

Server server;

User user;

Thing thing;

Gateway gateway;

// Setup phase: Server generates public and private keys

setup(pairing, &server);

// Shamir Secret Sharing: Distribute shared secrets

shamir\_secret\_sharing(pairing, &server, &gateway, 2, 3); // Threshold = 2, Number of shares = 3

// Register user, thing, and gateway with the server

user\_registration(pairing, &user, &server);

thing\_registration(pairing, &thing, &server);

gateway\_registration(pairing, &gateway, &server);

// Mutual Authentication: User-Gateway, Thing-Gateway, and Server-Gateway

mutual\_authentication\_user\_gateway(pairing, &user, &gateway);

mutual\_authentication\_thing\_gateway(pairing, &thing, &gateway);

mutual\_authentication\_server\_gateway(pairing, &server, &gateway);

// Clear elements and pairing

element\_clear(user.private\_key);

element\_clear(user.public\_key);

element\_clear(thing.private\_key);

element\_clear(thing.public\_key);

element\_clear(server.secret\_key);

element\_clear(server.public\_key);

element\_clear(gateway.shared\_key);

pairing\_clear(pairing);

return 0;

}

Key Additions for Mutual Authentication:

1. mutual\_authentication\_user\_gateway: Ensures that both the User and Gateway authenticate each other.
2. mutual\_authentication\_thing\_gateway: Both the Thing (IoT device) and Gateway authenticate each other.
3. mutual\_authentication\_server\_gateway: Both the Server and Gateway authenticate each other.
4. Each mutual authentication step uses pairing-based operations to ensure security.

Steps in Mutual Authentication:

* Each entity computes the pairing of the other’s public key with its private key.
* If both computed values match, mutual authentication succeeds.
* Mutual authentication occurs in both directions: between user-server, thing-gateway, and server-gateway.