Pedersen Commitment + Zero-Knowledge Authentication

Mathematical Foundations and Implementation

Mohamed Gamal **(ID: 10004437)**
AbdelRahman Ahmed **(ID: 10004269)**

December 22, 2024

1 Introduction

In this project, we implement a **Pedersen-commitment**-based authentication system using a **Schnorr-like Zero-Knowledge Proof (ZKP)**. This system ensures:

- Passwords are *never* transmitted in plaintext over the network.
- The server stores only a **commitment**, not the raw password.
- Authentication is performed through a ZKP, ensuring the client proves knowledge of the password without revealing it.
- Efficient measurements of computational cost, including CPU, RAM usage, and network latency.

2 Mathematics Behind Pedersen Commitment

2.1 Discrete Logarithm Problem

The Pedersen commitment scheme relies on the hardness of the *discrete logarithm problem*. Given a generator g of a group $\mathbb{Z}_P^*$ (with prime P), it is computationally infeasible to find x such that:

$$g^x \mod P = y,$$

if P is sufficiently large (e.g., 2048+ bits).

2.2 Pedersen Commitment Formula

A Pedersen commitment to a password p (integer) and a random secret r is computed as:

$$C = g^p \cdot h^r \mod P,$$

where g and h are distinct generators of the same group. The scheme guarantees two key properties:

- **Hiding Property**: Due to the inclusion of the random r, an adversary cannot deduce p from C.
- **Binding Property**: Once C is published, it is computationally infeasible for the committer to change the values of p or r without altering C.

These properties form the foundation of our authentication scheme.

3 System Workflow

3.1 Enrollment (Sign-Up)

The client performs the following steps during user registration:

1. Picks a password p and a random value r.
2. Computes the Pedersen commitment:

$$C = g^p \cdot h^r \mod P.$$

3. Sends $(\mathit{username}, C)$ to the server.
4. The server stores $\mathit{username} \mapsto C$ in its database.

3.2 Authentication (Login) via Zero-Knowledge Proof

To prove knowledge of p and r without revealing them, the client:

1. Picks fresh random values α and β.
2. Computes:

$$A = g^\alpha \cdot h^\beta \mod P.$$

3. Forms a challenge:

$$e = \mathrm{Hash}(C \| A).$$

4. Computes responses:

$$s_1 = \alpha + e \cdot p \mod (P-1), \quad s_2 = \beta + e \cdot r \mod (P-1).$$

5. Sends (A, s_1, s_2) to the server.
6. The server re-computes e and verifies:

$$g^{s_1} \cdot h^{s_2} \stackrel{?}{=} A \cdot C^e \mod P.$$

7. If the equality holds, authentication succeeds.

4 Implementation Details

4.1 Server Implementation

The server handles both sign-up and login requests. During login, it verifies the ZKP provided by the client. Key steps include:

Listing 1: Server `handleConnection`

```
func handleConnection(conn net.Conn) {
    // Parse incoming JSON for operation type.
    if operation == "SIGN_UP" {
        // Store {username -> commitment} mapping.
    } else if operation == "LOGIN" {
```

```
        // Retrieve commitment for username.
        // Verify ZKP:
        //   g^s1 * h^s2 ?= A * (C^e)
    }
}
```

4.2 Client Implementation

The client computes commitments during sign-up and generates ZKPs during login. Key steps include:

Listing 2: Client `login`

```
func login(username string) {
    // Retrieve stored password (p) and random secret (r).
    // Compute ephemeral proof values A, s1, s2.
    // Send (username, A, s1, s2) to server.
}
```

5 Efficiency Metrics

5.1 CPU and RAM Utilization

- **Server**: Measures memory allocation and CPU usage during ZKP verification using `runtime.MemStats`.
- **Client**: Profiles the cost of generating commitments and ZKPs.

5.2 Network Latency

- **Round-Trip Time**: Measures total time from request initiation to response reception.
- **Processing Time**: Captures server-side verification time.

5.3 Example Observations

- **Server Memory Usage**: +8 KB during sign-up, +4 KB during login.
- **Round-Trip Time**: *Sign-Up:* 1.2 ms, *Login:* 1.0 ms.

6 Security Considerations

- **Password Hiding**: Passwords are never transmitted or stored in plaintext.
- **Binding**: Commitments are immutable once created, ensuring integrity.
- **Replay Protection**: Fresh randomness (α, β) prevents reuse of ZKP.

7 Conclusion and Future Enhancements

This project demonstrates a privacy-preserving authentication scheme that ensures:

- Secure password handling using Pedersen commitments.
- Zero-Knowledge authentication via Schnorr-like proofs.
- Efficiency in computation and communication.

Future enhancements include:

- Adopting elliptic curve cryptography for improved performance.
- Incorporating secure channels (e.g., TLS) for network communication.
- Enhancing client-side secret management with OS-level secure storage.