CS371 Pong Project Writeup

Shubhanshu Pokharel, Aaron Lin, Ayham Yousef

November 2025

1 Background

We built a real-time multiplayer Pong game where two players control paddles on opposite sides of the screen, bouncing a ball back and forth. The first player to score five points wins, and players can choose to rematch or exit after each game. The implementation uses Python with Pygame for graphics and runs at 60 frames per second to ensure smooth gameplay.

The architecture follows a client-server model where the server maintains the authoritative game state and handles all physics calculations including ball movement, collisions, and scoring. Clients send their paddle positions and receive updated game state 60 times per second. When a player connects, they wait in a matchmaking queue until a second player joins, at which point the server starts the game. We also included a web-based leaderboard to track player statistics.

The original implementation had no security; anyone could connect and all data transmitted in plain text. We added user authentication with password hashing, session tokens that expire after 10 minutes, and encrypted all network communication using Fernet symmetric encryption. The main challenge was adding these security features without impacting the 60 FPS performance requirement or introducing noticeable lag.

Throughout development, we used Git as our source control. Everything can be found on our GitHub: `https://github.com/Sixteen1-6/Pong`

2 Design

When players connect, the server assigns them to left or right paddles and starts the game. The server calculates ball movement, detects collisions with walls and paddles, and tracks scoring. Every frame, clients send their paddle position and receive the complete game state back, ensuring both players see the same thing.

The synchronization happens through JSON messages exchanged 60 times per second. Clients include their paddle position and a sync counter, while the server responds with both paddle positions, ball location, and current score. This keeps the game synchronized even over network connections with some

latency. When a player scores five points, the game enters a "game over" state where both clients can vote to play again or quit.

For security, we added a separate authentication server on port 8081 that handles user registration and login. Passwords are hashed with SHA-256 and a unique random salt for each user. When someone logs in successfully, the server generates a cryptographically secure token that expires after 10 minutes. Players use this token to connect to the game server, which verifies it before allowing them to play.

All network communication is encrypted using Fernet, which provides AES-128 encryption with message authentication. We chose symmetric encryption because it's fast enough for real-time games. Each message takes about 1 millisecond to encrypt or decrypt. The game server and authentication server run independently, so login issues don't affect people already playing.

We organized the security code into four separate modules: user database for password hashing, token manager for session handling, encryption wrapper for Fernet, and the authentication server. This made it easier to test each piece independently and meant we could change security approaches without touching the game code.

3 Implementation

The server runs three services on different ports: authentication on **8081**, the game on **8080**, and a leaderboard website on port **80**. The authentication server handles each request in its own thread, while the game server spawns two threads per match to manage each player's connection. When the first player connects to the game server, they wait until a second player joins, then both game threads begin.

Matchmaking and threading follow this structure:

```
while server_running:
    client = accept_connection()
    token = receive_and_decrypt(client)

    if verify_token(token):
        if waiting_player exists:
            start_game_threads(waiting_player, client)
            waiting_player = None
        else:
            waiting_player = client
    else:
        send_encrypted(client, "INVALID_TOKEN")
        close(client)
```

Each thread maintains the full game state and synchronizes with the other. The left player's thread calculates ball physics and collisions, while both threads

process paddle updates. At **60 FPS**, threads exchange encrypted JSON messages with their clients containing paddle positions, ball coordinates, and scores. The server merges the states and sends an authoritative synchronized version to clients.

The main synchronization loop is:

```
while game_active:
    client_data = receive_and_decrypt(client)
    paddle_pos = client_data['paddle']

    update_ball_physics()
    check_collisions()
    update_score()

    game_state = {
        'left_paddle': left_player.paddle,
        'right_paddle': right_player.paddle,
        'ball': ball.position,
        'score': [left_score, right_score],
        'game_over': score >= 5
    }

    send_encrypted(client, game_state)
    sleep(1/60)  # Maintain 60 FPS
```

Security is implemented using wrapper functions. Passwords are stored using SHA-256 hashing with a random **16-byte salt**. Tokens are generated using Python's `secrets` module as **32-byte secure random values** and stored with expiration timestamps. Fernet encryption is used to convert JSON messages to encrypted bytes before transmission and decrypt received messages back into JSON.

Password hashing and token handling are structured as:

```
# Registration
function register_user(username, password):
    salt = generate_random_bytes(16)
    hash = SHA256(password + salt)
    store_user(username, hash, salt)

# Login
function login_user(username, password):
    user = load_user(username)
    hash = SHA256(password + user.salt)

    if hash == user.stored_hash:
        token = generate_secure_random(32)
```

```
        store_token(token, username, current_time + 600)
        return token
    else:
        return None
```

Client authentication begins with connecting to port 8081 and sending encrypted credentials. If valid, the server generates and returns a token. The client then connects to port 8080 and sends the token to gain access. Only after a valid acknowledgement does the game begin.

Client flow:

```
# Client authentication
connect_to_server(auth_server, 8081)
credentials = {'action': 'login', 'username': user, 'password': pass}
send_encrypted(credentials)
response = receive_and_decrypt()
token = response['token']

# Game connection
connect_to_server(game_server, 8080)
send_encrypted(token)
ack = receive_and_decrypt()

if ack == "TOKEN_OK":
    game_config = receive_and_decrypt()
    start_game_loop()
```

On the client side, Tkinter was used to create GUI login and registration screens before launching the Pygame window. After authentication, the client stores the token and uses it for all game communication. All game messaging is encrypted, while game logic such as paddle movement and ball collisions remains unchanged.

The encryption wrapper functions used throughout the system are:

```
function send_encrypted(socket, data):
    json_string = convert_to_json(data)
    encrypted_bytes = fernet.encrypt(json_string)
    socket.send(encrypted_bytes)

function receive_and_decrypt(socket):
    encrypted_bytes = socket.receive(2048)
    json_string = fernet.decrypt(encrypted_bytes)
    data = parse_json(json_string)
    return data
```

4 Challenges

Our biggest challenge was keeping the game running smoothly while adding security. We initially tried using RSA encryption because it's more secure, but it was way too slow. It would take $5 - 10$ milliseconds per message. At 60 FPS with messages going both ways, that would add over a second of lag per second, which made the game completely unplayable. We switched to Fernet symmetric encryption which only takes $1 - 2$ milliseconds per message. It's technically less secure than RSA since both sides share the same key, but for a game it's a reasonable trade-off.

We ran into weird issues with encrypted messages getting cut off randomly. The problem was that TCP doesn't guarantee message boundaries; sometimes one message gets split across multiple receives, or multiple messages arrive together. When we got half an encrypted message, decryption would fail with an error. Increasing the buffer size from 1024 to 2048 bytes helped a lot, reducing failures to about 1% of messages. The proper fix would be sending the message length first, then reading exactly that many bytes, but we ran out of time to implement it everywhere.

5 Lessons Learned

The biggest security lesson, however, was to never store passwords in plain text-even during development. We nearly did this "just for testing" but decided to implement proper hashing from the start. It only took a few extra minutes to add SHA256 with salts, and it meant we didn't have to refactor everything later. Even if someone gets access to the database, they can't recover the actual passwords.

We learned that encryption also needs authentication. Our first attempt had just used basic AES encryption, but that doesn't prevent someone from modifying encrypted messages. If an attacker flips some bits in an encrypted score, they could change a 3 to a 5 without even knowing what they're changing. Using Fernet fixed this because it includes message authentication that detects any tampering. The moral is to employ high-level crypto libraries that do the right thing, rather than gluing together low-level functions ourselves.

TCP was more complicated than we had thought. We had assumed each send() would match exactly one recv() on the other end, but TCP can split messages apart or join them together. This sometimes resulted in truncation of our encrypted messages. The solution is proper framing of the messages, where you first send the length, followed by the data.

In terms of performance, we spent time researching the fastest encryption algorithms, since we assumed that would be our main performance problem. It turns out JSON serialization was actually slower than encryption. Using a faster JSON library yielded better results for us speedup than all our encryption optimization would have.

The modular design really paid off. Keeping security code separate from

game code meant we could test each piece independently and swap out encryption algorithms without touching the game. When debugging, we could immediately tell which module had issues based on the error type. The extra time spent planning the architecture upfront saved us debugging time later.

6 Conclusions

We successfully implemented authentication and encryption for multiplayer Pong while keeping gameplay at 60 FPS. The implementation added 556 lines of security code, modularized into four components, and involving only 70 lines of changes to existing game code. Performance overhead remained under 2ms per message, and the core game logic required no changes due to clean separation between security and gameplay concerns.

The modular architecture was extremely helpful during development. Independent security modules enabled isolated testing and allowed switching encryption algorithms without affecting game code. Separating the authentication server from the game server on different ports prevented authentication failures from disrupting active matches and enabled independent scaling. The design follows industry-standard practices, including SHA-256 password hashing with salts, generation of cryptographically secure tokens, and Authenticated Encryption using a combined AES-128 and HMAC scheme.

Key lessons learned include the importance of designing security from the beginning rather than retrofitting it later, the value of testing before optimizing, and understanding TCP is a byte stream that needs appropriate framing of messages. The implementation shows that security and performance are not in conflict when appropriate algorithms are chosen and effective integration is emphasized.