HW06: Secure Peer-to-Peer Rock-Paper-Scissors

Simone Nolé
1940213

December 4, 2025

Abstract

This report presents the design and implementation of a secure Peer-to-Peer (P2P) protocol for playing "Rock, Paper, Scissors". To ensure fairness and prevent cheating in a trustless environment, the protocol implements a **Bit Commitment Scheme** using SHA-256 hashing to hide moves, and strict **Timeout Mechanisms** to handle disconnection attempts by losing players. The system is deployed using Docker containers to simulate distinct network nodes.

Contents

1 Introduction

The objective of this assignment is to allow two remote entities, Alice and Bob, to play Rock-Paper-Scissors over a network. Unlike a Client-Server architecture where a central authority guarantees fairness, this solution adopts a **Peer-to-Peer** approach.

In a P2P scenario, neither player trusts the other. This introduces specific security challenges:

- **The Pre-knowledge Attack:** If Alice sends her move first in cleartext, Bob can see it and choose the winning move.
- **The Abort Attack (Sore Loser):** If Alice waits for Bob to reveal his move and realizes she has lost, she might disconnect to avoid the loss being recorded.

2 Protocol Design: The Commitment Scheme

To address the "Pre-knowledge Attack", we implemented a cryptographic **Commitment Scheme**. This ensures that a player chooses a move and "locks" it before revealing it, making it impossible to change later.

The protocol proceeds in three phases:

2.1 Phase 1: Commitment

Alice (the Initiator) selects her move M_A and generates a high-entropy random nonce N_A (128-bit). She calculates the commitment Hash H:

$$H = \text{SHA-256}(M_A || N_A)$$

Alice sends H to Bob. Bob now knows Alice has played, but cannot know what she played.

2.2 Phase 2: The Game (Bob's Move)

Bob (the Peer/Listener), upon receiving H, sends his move M_B in cleartext to Alice.

- Bob cannot cheat because he doesn't know M_A.
- Alice cannot change M_A because it must match the hash H she already sent.

2.3 Phase 3: Reveal and Verification

Upon receiving M_B, Alice sends the pair (M_A, N_A) to Bob. Bob verifies the game validity by re-computing the hash:

$$H_{check} = \text{SHA-256}(M_A || N_A)$$

If $H_{check} == H$, the game is valid, and the winner is determined.

3 Security Implementation Details

3.1 Preventing Brute Force (High Entropy Nonce)

A simple hash of the move (e.g., SHA-256("Rock")) is vulnerable to brute-force attacks since the search space is only size 3. To prevent Bob from guessing Alice's move, we use the `secrets` library to generate a cryptographically strong nonce.

```
import secrets
import hashlib

# 16 bytes = 128 bits of entropy. Impossible to brute-force.
nonce = secrets.token_hex(16)
commitment = hashlib.sha256((move + nonce).encode()).hexdigest()
```

Listing 1: Generating the Commitment

3.2 Preventing Abort Attacks (Timeout Strategy)

The "Abort Attack" occurs when Alice receives Bob's move, calculates the result locally, sees she has lost, and refuses to send the Reveal message (Phase 3). To solve this, Bob implements a strict **Socket Timeout**. This rule ensures that disconnecting is equivalent to losing.

4 Infrastructure and Virtualization

To meet the requirement of implementing the protocol on distinct virtual machines and simulating a realistic network environment, it has been adopted a **Container-based architecture** using Docker.

This approach ensures process isolation and provides a separate network stack for each actor (Alice and Bob), effectively simulating two distinct machines connected via a local network.

4.1 Docker Build Environment

The application environment for each service (`alice`, and `bob`) is defined by a single `Dockerfile`.

```
FROM python:3.10-slim
WORKDIR /app
COPY . /app
```

Listing 2: Dockerfile

4.2 Docker Compose Configuration

The orchestration of the environment is managed via `docker-compose`, which defines three services:

- **Bob:** It is accessible to alice via the hostname `bob`.
- **Alice:** This executes the client logic.

The following is the configuration used to deploy the environment:

```
services:
  bob:
    build: .
    command: python bob.py
    ports:
      - "8080:8080"
    hostname: bob
    restart: always
  alice:
    build: .
    command: python alice.py
    depends_on:
      - bob
    stdin_open: true
    tty: true
    restart: always
```

Listing 3: Docker Compose Configuration

4.3 Network Simulation

Docker automatically creates a virtual bridge network (default subnet). Within this network, **Service Discovery** is handled via internal DNS. Instead of hardcoding IP addresses (e.g., `192.168.1.x`), the alice.py connects to the server using the hostname `bob`. This abstraction mimics a real-world DNS resolution scenario and makes the code environment-agnostic.

- **Server Address:** `bob` (Resolved automatically by Docker DNS)
- **Port:** 8080 (Mapped for internal communication)

5 Execution Logs

Below are the execution traces proving the fairness of the protocol.

5.1 Scenario A: Honest Game

Alice plays Paper (hidden), Bob plays Scissors. Alice reveals Rock. Bob wins.

```
# Alice's Output
[Alice] I'm going to meet Bob...
[Alice] Arrivede to Bob.
[Alice] Waiting for message...  w Enable Watch
[Alice] Sent my move (Paper) without nonce (62345b125040d8a9fc9e340abfed34bd), I want to
    be sure that Bob cannot cheat
[Alice] Waiting for message...
[Alice] Received Bob's move: Scissors
[Alice] My commitment: 0cabbbd671b5542e61a1fb8b676670b5898a0f4aa4333fedca66e7fd79faf6e3
[Alice] Commitment verified successfully!
[Alice] Determining winner: Me(paper) vs Bob(scissors)
[Alice] The winner is: Bob
[Alice] Should I send the nonce to Bob? I can escape...(yes/no)
yes
[alice] Sent nonce to Bob.
[Alice] game over, connection closed.

# Bob's output
```

```
[BOB] Waiting for Alice to arrive...
[BOB] Alice has arrived.
[BOB] Sent game start message to Alice.
[BOB] Waiting for message...
[BOB] Received Alice's commitment: 0
    cabbbd671b5542e61a1fb8b676670b5898a0f4aa4333fedca66e7fd79faf6e3
[BOB] My move is: Scissors
[BOB] Sent my move to Alice.
[BOB] Waiting 5 seconds for Alice to reveal nonce...
[BOB] Waiting for message...
[BOB] Received Alice's nonce: 62345b125040d8a9fc9e340abfed34bd
[BOB] Alice move: Paper
[Bob] Determining winner: Alice(paper) vs Me(scissors)
[BOB] The winner is: Me
[BOB] Waiting for message...
[Bob] game over. Connection closed.
```

Listing 4: Honest Game Log

5.2 Scenario B: Abort Attack (Cheating Attempt)

Alice plays Rock, Bob plays Paper. Alice sees she lost and kills the connection to avoid admitting defeat. Bob wins by timeout.

```
# Alice's output
[Alice] I'm going to meet Bob...
[Alice] Arrivede to Bob.
[Alice] Waiting for message...  w Enable Watch
[Alice] Sent my move (Rock) without nonce (ef00291f1d4e3141f00939c7632cf142), I want to
    be sure that Bob cannot cheat
[Alice] Waiting for message...
[Alice] Received Bob's move: Paper
[Alice] My commitment: de743c777e236789b2a87bc1e01de8e885d4f79f52b4b53766699d42073fd8ef
Alice] Commitment verified successfully!
[Alice] Determining winner: Me(rock) vs Bob(paper)
[Alice] The winner is: Bob
[Alice] Should I send the nonce to Bob? I can escape...(yes/no)
no
[alice] I chose not to send the nonce to Bob. I have to run!
[Alice] game over, connection closed.

# Bob's output
[BOB] Waiting for Alice to arrive...
[BOB] Alice has arrived.
[BOB] Sent game start message to Alice.
[BOB] Waiting for message...
[BOB] Received Alice's commitment:
    de743c777e236789b2a87bc1e01de8e885d4f79f52b4b53766699d42073fd8ef
[BOB] My move is: Paper
[BOB] Sent my move to Alice.
[BOB] Waiting 5 seconds for Alice to reveal nonce...
[BOB] Waiting for message...
[Bob] Alice ran away! I win by default!
[Bob] game over. Connection closed.
```

Listing 5: Cheating Attempt Log

6 Conclusion

The implemented Peer-to-Peer protocol successfully mitigates both information leakage (via Bit Commitment) and evasion tactics (via Timeouts). This demonstrates that a fair game can be conducted over a network without relying on a central server, provided that the protocol enforces strict rules on information exchange and timing.

7 Code

7.1 tcp_json.py

```python
import json
import struct

def recvall(sock, n):
    """Helper function to receive EXACTLY n bytes or die trying"""
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None # the connection was closed
        data += packet
    return data

def receive_json(sock):

    header = recvall(sock, 4)
    if not header:
        return None # the connection was closed

    msg_length = struct.unpack('>I', header)[0]

    payload_bytes = recvall(sock, msg_length)

    try:
        packet_dict = json.loads(payload_bytes.decode('utf-8'))
        return packet_dict
    except json.JSONDecodeError:
        print("ERROR: Malformed JSON received!")
        return None

def send_json(sock, packet_dict):

    json_str = json.dumps(packet_dict)

    data_bytes = json_str.encode('utf-8')

    msg_length = len(data_bytes)

    header = struct.pack('>I', msg_length)

    sock.sendall(header + data_bytes)
```

Listing 6: tcp_json.py code that wrappers sendall and recvall function for TCP sockets and manages JSON communications properly

7.2 bob.py

```python
import json
import struct

def recvall(sock, n):
    """Helper function to receive EXACTLY n bytes or die trying"""
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None # the connection was closed
        data += packet
    return data

def receive_json(sock):

    header = recvall(sock, 4)
    if not header:
        return None # the connection was closed
```

```
19
20      msg_length = struct.unpack('>I', header)[0]
21
22      payload_bytes = recvall(sock, msg_length)
23
24      try:
25          packet_dict = json.loads(payload_bytes.decode('utf-8'))
26          return packet_dict
27      except json.JSONDecodeError:
28          print("ERROR: Malformed JSON received!")
29          return None
30
31  def send_json(sock, packet_dict):
32
33      json_str = json.dumps(packet_dict)
34
35      data_bytes = json_str.encode('utf-8')
36
37      msg_length = len(data_bytes)
38
39      header = struct.pack('>I', msg_length)
40
41      sock.sendall(header + data_bytes)
```

Listing 7: bob.py is the server side of this Peer-To-Peer connection

7.3 alice.py.py

```
1   import secrets
2   import socket
3   from hashlib import sha256
4
5   from time import sleep
6   from tcp_json import send_json
7   from tcp_json import receive_json
8
9   HOST = 'bob'
10  PORT = 8080
11
12  my_move = ""
13  my_nonce = ""
14  bob_move = ""
15
16  def determine_winner(my_move, bob_move):
17      a = my_move.lower()
18      b = bob_move.lower()
19      print(f"[Alice] Determining winner: Me({a}) vs Bob({b})")  # Debug
20
21      valid_moves = ["rock", "paper", "scissors"]
22
23      if a not in valid_moves or b not in valid_moves:
24          return "Error: Invalid Move"
25
26      if a == b:
27          return "Draw"
28
29      if (a == "rock" and b == "scissors") or \
30         (a == "scissors" and b == "paper") or \
31         (a == "paper" and b == "rock"):
32          return "Me"
33
34      return "Bob"
35
36  def handle_bob_move(message, conn):
37      global bob_move
38      bob_move = message.get("value")
39      print(f"[Alice] Received Bob's move: {bob_move}")
40
41      commitment_check = sha256((my_move + my_nonce).encode()).hexdigest()
42      print(f"[Alice] My commitment: {commitment_check}")
43
```

```
    print("[Alice] Commitment verified successfully!")
    winner = determine_winner(my_move, bob_move)
    print(f"[Alice] The winner is: {winner}")
    if winner == "Bob":
        print("[Alice] Should I send the nonce to Bob? I can escape...(yes/no)")
        choice = input().strip().lower()
        if choice == "yes":
            response = {
                "type": "reveal-nonce",
                "value": my_nonce
            }
            send_json(conn, response)
            print("[alice] Sent nonce to Bob.")
        else:
            print("[alice] I chose not to send the nonce to Bob. I have to run!")
    else:
        response = {
            "type": "reveal-nonce",
            "value": my_nonce
        }
        send_json(conn, response)
        print("[alice] Sent nonce to Bob.")
    return False


def game(message, conn):
    global my_move, my_nonce
    my_move = secrets.choice(["Rock", "Paper", "Scissors"])
    my_nonce = secrets.token_hex(16)
    commitment = sha256((my_move + my_nonce).encode()).hexdigest()
    message = {
        "type" : "game-commitment",
        "value" : commitment
    }

    send_json(conn, message)
    print(f"[Alice] Sent my move ({my_move}) without nonce ({my_nonce}), I want to be
    sure that Bob cannot cheat")
    return True


def handle(message, conn):
    msg_type = message.get("type")
    match msg_type:
        case "game":
            return game(message, conn)
        case "bob-move":
            return handle_bob_move(message, conn)


def main():

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as conn:
        print("[Alice] I'm going to meet Bob...")
        sleep(2)
        conn.connect((HOST, PORT))
        conn.settimeout(5.0)
        print("[Alice] Arrivede to Bob.")

        while True:
            print("[Alice] Waiting for message...")
            # Do not pass 0 here; pass no message (or None) so the function reads from
    socket
            message = receive_json(conn)
            if not message:
                print("[Alice] No message received, closing connection")
                break
            #print(f"message received : {message}")
            if handle(message, conn) == False:
                break
        conn.close()
        print("[Alice] game over, connection closed.")

```

```
115
116 if __name__ == "__main__":
117     main()
```

Listing 8: alice.py is like the client side of the Peer-To-peer connection