HW07: Secure Distributed Dice Game

Course: Cybersecurity

Jacopo Rossi
M.Sc. in Engineering in Computer Science and Artificial Intelligence
Sapienza University of Rome

December 11, 2025

Abstract

This report details the design and implementation of a secure, peer-to-peer dice game played between two untrusted parties, Alice and Bob. The system is containerized using Docker to simulate two distinct virtual machines communicating over a TCP network. To ensure fairness without a Trusted Third Party (TTP), a cryptographic *Commit-then-Reveal* protocol is employed. This ensures that the outcome of the dice rolls is derived from shared entropy ($N_A \oplus N_B$) and that neither party can manipulate the result once the commitment phase is complete.

Code Availability: The complete source code, including Docker configuration and usage instructions, is available on GitHub at:
`https://github.com/enoughpaladin00/secure-dice-game`

1 Introduction

The objective of this assignment is to simulate a match of dice between two players. A match consists of N rounds, where in each round k dice are rolled simultaneously. The player with the highest sum wins the round. The core challenge is security in a distributed environment: preventing "Bit-Fixing" or "Look-Ahead" cheating, where a player waits to see the opponent's move before deciding their own to force a favorable outcome.

2 Protocol Design

2.1 Cryptographic Primitives

To secure the game, we utilize two main concepts:

- **SHA-256 Hashing:** Used as a commitment scheme. Since $H(x)$ is a one-way function, sending $H(x)$ commits a player to value x without revealing it.
- **XOR (Exclusive OR):** Used to mix the random nonces of both players. The property $S = A \oplus B$ guarantees that if at least one input is uniformly random, the output S is uniformly random.

2.2 The Protocol Flow

For every round of the game, the following steps are executed:

1. **Generation:** Alice generates a cryptographically secure random nonce R_A. Bob generates R_B.
2. **Commitment:**

$$C_A = \text{SHA256}(R_A), \quad C_B = \text{SHA256}(R_B)$$

   Alice and Bob exchange these hash values (C_A, C_B).

3. **Reveal:** Once commitments are exchanged, players exchange the actual nonces R_A and R_B.
4. **Verification:** Alice computes $H(R_{B\_recv})$ and compares it to C_B. Bob does the same for Alice. If the hashes do not match, the protocol aborts (Cheating Detected).
5. **Derivation:** A shared seed S is calculated:

$$S = R_A \oplus R_B$$

   The dice rolls are generated deterministically using a PRNG seeded with S. To ensure distinct rolls for both players, we use offset seeds: $S_{Alice} = S+0$, $S_{Bob} = S+1$.

3 System Implementation

3.1 Architecture

The system is implemented in Python and orchestrated via Docker Compose.

- **Alice (Server):** Acts as the game initiator. She listens on TCP port 65432 and defines the game parameters (k dice, n rounds).
- **Bob (Client):** Connects to Alice. We implemented a **robust retry mechanism**: since Alice requires manual input to start, Bob will repeatedly attempt to connect every 2 seconds until Alice's server is ready, preventing race conditions or crashes on startup.

3.2 Key Code Snippets

1. Secure Nonce Generation: We rely on the OS entropy pool ('os.urandom') rather than standard random libraries to ensure unpredictability.

```python
def generate_nonce(length=32):
    """Generates a cryptographically secure random number using OS entropy.
    """
    return os.urandom(length)
```

Listing 1: Secure Random Generation

2. Verification and Result Derivation: This snippet demonstrates the integrity check and the XOR combination.

```python
# Verify that the revealed nonce matches the previous commitment
calculated_hash = hashlib.sha256(opponent_nonce).hexdigest()
if calculated_hash != opponent_commitment:
    raise ValueError("Cheating Detected! Hash mismatch.")

# XOR to derive shared randomness
shared_seed = bytes(a ^ b for a, b in zip(my_nonce, opponent_nonce))
```

Listing 2: Verification Logic

4 Execution Manual

The environment simulates two separate machines using a Docker Bridge Network. Below are the commands to reproduce the experiment.

4.1 1. Start the Environment

Build and start the containers in detached mode.

```
$ docker-compose up -d --build
```

4.2 2. Configure Alice (Server)

Alice's container requires interaction to set the game rules. We attach to her TTY:

```
$ docker attach alice_vm
```

Input: The system will prompt for the number of dice and rounds.

```
Enter number of dice (k): 3
Enter number of rounds: 5
```

4.3 3. Observe Bob (Client)

In a separate terminal window, we view Bob's logs. Thanks to the retry mechanism, Bob waits for Alice to finish configuration and then connects automatically.

```
$ docker logs -f bob_vm
```

4.4 4. Shutdown

To stop and remove the containers:

```
$ docker-compose down
```

5 Security Analysis

Why cheating is impossible:

- **Changing the Roll:** If Alice tries to change her nonce R_A after seeing Bob's value, the hash of the new nonce will not match the commitment C_A she sent earlier. Bob's verification step will fail.
- **Forcing a Result:** Even if Bob tries to pick a specific R_B to influence the XOR sum, he cannot know R_A at the time of commitment (due to the Pre-image resistance of SHA-256). Therefore, as long as Alice's nonce is uniformly random, the result $R_A \oplus R_B$ remains uniformly random.

6 Conclusion

The implemented solution satisfies all requirements: it runs on isolated virtual environments (Docker), allows customizable game parameters, and guarantees game fairness through a mathematically secure Commit-then-Reveal protocol.