Homework 07: Secure Dice Game Protocol

Nicolas Leone
Student ID: 1986354
Cybersecurity

December 8, 2025

Contents

1 Introduction

Dice games are ancient games of chance where players roll dice and compare results to determine a winner. When Alice and Bob want to play a dice game remotely over a network, several challenges arise similar to those in other distributed games.

In a physical dice game, both players roll their dice simultaneously and reveal results together. However, over a network, perfect simultaneity is impossible due to network latency and sequential communication. This creates opportunities for cheating:

1. **Sequential Communication**: One player must send their result first. The second player can cheat by adjusting their reported result after seeing the first player's roll.
2. **Result Verification**: Without witnessing the actual dice roll, how can we verify that a player honestly reported their dice results?
3. **Trust Issues**: Without a trusted third party, how can we ensure neither player cheats?

A cryptographic commitment scheme solves these problems by allowing Alice to "commit" to her dice roll result without revealing it. The protocol uses a two-phase approach:

- **Commit Phase**: Alice rolls her dice, computes the sum, and sends a cryptographic hash of the sum. Bob cannot determine Alice's sum from the hash (hiding property).
- **Reveal Phase**: After Bob rolls and sends his result, Alice reveals her original dice and sum, proving they match the commitment. Alice cannot change her result (binding property).

1.1 Assignment Requirements

This homework addresses the following specific requirements:

1. **Dice Game with k Dice**: Allow Alice and Bob to play with k six-sided dice (configurable number).
2. **Game Mechanics**: A single game consists of simultaneous rolling of all dice by each player. Each player sums their results and compares them. The winner is the player with the highest total.
3. **Match System**: A match is a sequence of games of predefined length. The match winner is determined by who wins more individual games.
4. **Security Against Cheating**: Ensure all operations are secure against possible cheating by Alice and/or Bob using cryptographic mechanisms.
5. **Design Choices Documentation**: Report all design choices and implementation decisions.
6. **Double Requirement (Virtual Machines)**: Create two virtual machines that implement the protocol, specifying all code and commands. This is satisfied using Docker containers providing OS-level virtualization.

1.2 Solution Overview

Our solution employs:

- **Hash-based commit-reveal protocol** using SHA-256 for cryptographic commitments
- **TCP socket communication** for network protocol implementation
- **Docker containers** for virtual machine deployment (two isolated environments: Alice's client and Bob's server)
- **Python 3.11** for implementation with cryptographic libraries
- **Match tracking system** for multi-game matches with score accumulation

2 Cryptographic Background

2.1 Commitment Schemes

A commitment scheme is a cryptographic primitive with two essential properties:

Hiding The commitment C does not reveal information about the committed value v:

$$P(\text{Adversary guesses } v \mid C) \approx \frac{1}{|\mathcal{V}|}$$

where $\mathcal{V}$ is the set of possible values.

Binding After committing to v, it is computationally infeasible to find another value $v' \neq v$ that produces the same commitment:

$$P(\text{Find } v' \neq v \text{ such that } \text{Commit}(v') = C) \approx 0$$

2.2 Hash-Based Commitment

Our implementation uses a hash-based commitment scheme with SHA-256:

$$C = \text{SHA-256}(s \,||\, r) \tag{1}$$

where s is the committed value (dice sum as string), r is a random nonce (256-bit cryptographically secure random value), $||$ denotes concatenation, and C is the commitment (256-bit hash).

Why use a nonce? The dice sum has a limited range. For k=3 dice, the sum ranges from 3 to 18 (only 16 possible values). Without a nonce, Bob could perform a dictionary attack:

1. For each possible sum $s \in \{3, 4, ..., 18\}$:
2. Compute $H_s = \text{SHA-256}(s)$
3. Compare C with all H_s to determine Alice's sum

This attack succeeds because the input space is small. With a 256-bit nonce, Bob must try approximately 16×2^{256} combinations, making the attack computationally infeasible. Even with current computational power (e.g., 10^{18} hashes/second), brute-forcing would take approximately 10^{59} years.

2.3 Security Properties

Property	Implementation
Hiding	SHA-256 is a one-way function; given C, computing s requires brute force over all possible sums and nonces
Binding	SHA-256 collision resistance: finding $s' \neq s$ with SHA-256$(s'\|\|r') = C$ is computationally infeasible
Nonce Security	256-bit nonce provides 2^{256} possible values per sum, making dictionary attacks infeasible
Deterministic Verification	Bob can verify by recomputing $C' =$ SHA-256$(s\|\|r)$ and comparing $C' = C$

Table 1: Security properties of the commitment scheme

3 Protocol Design

3.1 Protocol Flow

The protocol for a single game consists of four phases:

Phase 1: Commitment

1. Alice rolls k dice: $d_1, d_2, ..., d_k$ where each $d_i \in \{1, 2, 3, 4, 5, 6\}$
2. Alice computes sum: $s_A = \sum_{i=1}^{k} d_i$
3. Alice generates a cryptographically secure random nonce r (256 bits)
4. Alice computes the commitment: $C =$ SHA-256$(s_A||r)$
5. Alice sends C and k to Bob over the network

Phase 2: Bob's Roll

1. Bob receives the commitment C and number of dice k from Alice
2. Bob rolls k dice: $d'_1, d'_2, ..., d'_k$
3. Bob computes sum: $s_B = \sum_{i=1}^{k} d'_i$
4. Bob sends $(s_B, [d'_1, d'_2, ..., d'_k])$ to Alice

At this point:

- Bob has committed to s_B by sending it
- Alice is cryptographically bound to s_A (cannot change without breaking commitment)
- Bob cannot determine s_A from C (hiding property)

Phase 3: Reveal

1. Alice receives Bob's sum s_B and dice results
2. Alice sends $(s_A, [d_1, d_2, ..., d_k], r)$ to Bob

Phase 4: Verification and Result

1. Bob verifies dice sum: $\sum d_i = s_A$
2. Bob computes $C' = \text{SHA-256}(s_A||r)$
3. Bob verifies that $C' = C$
4. If verification fails, Alice cheated and the protocol aborts
5. If verification succeeds, Bob determines the winner:
   - If $s_A > s_B$: Alice wins
   - If $s_B > s_A$: Bob wins
   - If $s_A = s_B$: Tie
6. Bob sends the result to Alice

3.2 Match Protocol

A match consists of multiple games:

1. Initialize counters: $W_A = 0$ (Alice wins), $W_B = 0$ (Bob wins), $T = 0$ (ties)
2. For each game $g = 1$ to n (match length):
   - Execute the four-phase game protocol
   - Update counters based on game winner
3. Determine match winner:
   - If $W_A > W_B$: Alice wins the match
   - If $W_B > W_A$: Bob wins the match
   - If $W_A = W_B$: Match tied

3.3 Message Format

All messages use JSON encoding transmitted over TCP sockets with newline delimiters. Each message has the structure:

```
{
    "type": "MESSAGE_TYPE",
    "data": {
        "field1": "value1",
        "field2": "value2"
    }
}
```

Listing 1: Message structure

Message types and their data fields:

Type	Direction	Data Fields
COMMIT	Alice → Bob	`commitment`: 64-char hex (SHA-256), `num_dice`: integer
RESULT	Bob → Alice	`bob_sum`: integer, `bob_dice`: array of integers
REVEAL	Alice → Bob	`alice_sum`: integer, `alice_dice`: array, `nonce`: 64-char hex
MATCH_RESULT	Bob → Alice	`winner`: 0/1/2, `message`: string, sums and dice
ERROR	Either	`message`: error description

Table 2: Protocol message types

4 Implementation

4.1 Architecture Overview

The implementation consists of three main components:

- **shared/protocol.py**: Common protocol logic, commitment scheme, and dice game rules (220 lines)
- **alice/alice.py**: Client implementation (Alice's side) (220 lines)
- **bob/bob.py**: Server implementation (Bob's side) (240 lines)

Both Alice and Bob run in separate Docker containers connected via a Docker network, simulating two virtual machines communicating over TCP/IP.

4.2 Commitment Scheme Implementation

```python
import hashlib
import secrets

class CommitmentScheme:
    @staticmethod
    def generate_nonce(length=32):
        """Generate cryptographically secure random nonce."""
        return secrets.token_hex(length)

    @staticmethod
    def commit(value, nonce):
        """Create SHA-256 commitment."""
        data = f"{value}||{nonce}".encode('utf-8')
        return hashlib.sha256(data).hexdigest()

    @staticmethod
    def verify(commitment, value, nonce):
        """Verify revealed value matches commitment."""
```

```
        expected = CommitmentScheme.commit(value, nonce)
        return commitment == expected
```

Listing 2: Cryptographic commitment implementation

Key implementation details:

- `secrets.token_hex()`: Uses OS-provided entropy source (`/dev/urandom`) for cryptographically secure randomness
- `hashlib.sha256()`: NIST-approved hash function (FIPS 180-4)
- Separator `||`: Prevents ambiguity in concatenation
- Value is converted to string before hashing for consistent encoding

4.3 Dice Game Logic

```
import random

class DiceLogic:
    @staticmethod
    def roll_dice(num_dice=1):
        """Roll num_dice six-sided dice."""
        return [random.randint(1, 6) for _ in range(num_dice)]

    @staticmethod
    def calculate_sum(dice_results):
        """Calculate the sum of dice results."""
        return sum(dice_results)

    @staticmethod
    def determine_winner(sum1, sum2):
        """Determine winner based on dice sums."""
        if sum1 > sum2:
            return 1  # Player 1 wins
        elif sum2 > sum1:
            return 2  # Player 2 wins
        else:
            return 0  # Tie
```

Listing 3: Dice rolling and winner determination

Design Choice: We use Python's `random.randint(1, 6)` for dice rolls. While this uses a pseudorandom number generator (not cryptographically secure), it is appropriate for dice simulation as:

- The randomness is for game mechanics, not security
- Security is provided by the commitment scheme with cryptographic nonce
- Python's Mersenne Twister provides sufficient randomness for dice simulation

4.4 Network Communication

TCP sockets provide reliable, ordered message delivery. Messages are newline-delimited for parsing:

```python
def send_message(sock, msg_type, **kwargs):
    message = ProtocolMessage.create(msg_type, **kwargs)
    sock.sendall(message.encode('utf-8') + b'\n')

def receive_message(sock):
    data = b''
    while True:
        chunk = sock.recv(1)
        if not chunk:
            raise ConnectionError("Connection closed")
        if chunk == b'\n':
            break
        data += chunk
    return ProtocolMessage.parse(data.decode('utf-8'))
```

Listing 4: Message sending and receiving

Design Choice: We use newline-delimited JSON instead of fixed-length headers because:

- Simpler implementation and debugging
- Human-readable protocol for testing
- Small message sizes make efficiency less critical
- Sufficient for educational purposes

4.5 Alice's Implementation (Client)

Alice implements the commitment protocol from the client perspective:

```python
def play_game(self, game_num):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((self.server_host, self.server_port))

        # Phase 1: Roll and commit
        alice_dice = self.dice_logic.roll_dice(self.num_dice)
        alice_sum = self.dice_logic.calculate_sum(alice_dice)
        alice_nonce = self.commitment_scheme.generate_nonce()
        alice_commitment = self.commitment_scheme.commit(
            str(alice_sum), alice_nonce
        )
        send_message(sock, MSG_COMMIT,
                     commitment=alice_commitment,
                     num_dice=self.num_dice)

        # Phase 2: Receive Bob's result
        msg = receive_message(sock)
        bob_sum = msg['data']['bob_sum']
        bob_dice = msg['data']['bob_dice']

        # Phase 3: Reveal
```

```
        send_message(sock, MSG_REVEAL,
                     alice_sum=alice_sum,
                     alice_dice=alice_dice,
                     nonce=alice_nonce)

        # Phase 4: Receive game result
        msg = receive_message(sock)
        winner = msg['data']['winner']
        return (alice_sum, bob_sum, winner)
```

Listing 5: Alice's game protocol (excerpted)

Alice maintains match statistics across multiple games and displays final results after all games complete.

4.6 Bob's Implementation (Server)

Bob implements the server side, verifying Alice's commitment:

```
def handle_game(self, conn, addr, game_num):
    # Phase 1: Receive commitment
    msg = receive_message(conn)
    alice_commitment = msg['data']['commitment']
    num_dice = msg['data']['num_dice']

    # Phase 2: Roll and send result
    bob_dice = self.dice_logic.roll_dice(num_dice)
    bob_sum = self.dice_logic.calculate_sum(bob_dice)
    send_message(conn, MSG_RESULT,
                 bob_sum=bob_sum,
                 bob_dice=bob_dice)

    # Phase 3: Receive reveal
    msg = receive_message(conn)
    alice_sum = msg['data']['alice_sum']
    alice_dice = msg['data']['alice_dice']
    alice_nonce = msg['data']['nonce']

    # Phase 4: Verify
    # Check dice sum matches
    if alice_sum != sum(alice_dice):
        send_message(conn, MSG_ERROR,
                     message="Invalid sum!")
        return None

    # Check commitment
    is_valid = self.commitment_scheme.verify(
        alice_commitment, str(alice_sum), alice_nonce
    )
    if not is_valid:
        send_message(conn, MSG_ERROR,
                     message="Cheating detected!")
        return None

    # Determine winner
    winner = self.dice_logic.determine_winner(
        alice_sum, bob_sum
    )
```

```
    send_message(conn, MSG_MATCH_RESULT,
                 winner=winner, ...)
    return (alice_sum, bob_sum, winner)
```

Listing 6: Bob's verification protocol (excerpted)

Bob performs two verification steps:

1. **Sum verification**: Ensures reported sum matches actual dice values
2. **Commitment verification**: Ensures Alice didn't change her result after seeing Bob's roll

5 Docker Deployment: Virtual Machine Implementation

5.1 Solving the "Double" Requirement

The assignment requires creating **two virtual machines** that implement the protocol (the "Double" requirement). This has been achieved using **Docker containers**, which provide OS-level virtualization equivalent to traditional virtual machines for this use case.

Each Docker container represents an isolated virtual environment with:

- Separate filesystem and process isolation
- Independent network namespace
- Dedicated runtime environment (Python 3.11)
- Network communication via TCP/IP over a bridge network

This architecture simulates two distinct machines (Alice's client machine and Bob's server machine) connected over a network, satisfying the virtual machine requirement while being more lightweight and efficient than traditional hypervisor-based VMs.

5.2 Container Architecture

The deployment uses Docker Compose to orchestrate two containers. Alice's container runs the client, Bob's container runs the server, and they communicate via a Docker bridge network named `dice_network`.

5.3 Dockerfile for Bob (Server)

```
FROM python:3.11-slim

WORKDIR /app

# Copy shared protocol module
COPY shared/ /app/shared/

# Copy Bob's code
```

```
COPY bob/bob.py /app/

# Make script executable
RUN chmod +x /app/bob.py

# Expose game port
EXPOSE 5555

# Run Bob's server
CMD ["python", "/app/bob.py"]
```

Listing 7: Bob's Dockerfile

5.4 Dockerfile for Alice (Client)

```
FROM python:3.11-slim

WORKDIR /app

# Copy shared protocol module
COPY shared/ /app/shared/

# Copy Alice's code
COPY alice/alice.py /app/

# Make script executable
RUN chmod +x /app/alice.py

# Run Alice's client
CMD ["python", "/app/alice.py"]
```

Listing 8: Alice's Dockerfile

5.5 Docker Compose Configuration

```
services:
  bob:
    build:
      context: .
      dockerfile: bob/Dockerfile
    container_name: dice_bob
    hostname: bob
    networks:
      - game_network
    environment:
      - BOB_HOST=0.0.0.0
      - BOB_PORT=5555
      - NUM_GAMES=5
    ports:
      - "5555:5555"
```

```
  alice:
    build:
      context: .
      dockerfile: alice/Dockerfile
    container_name: dice_alice
    hostname: alice
    networks:
      - game_network
    environment:
      - BOB_HOST=bob
      - BOB_PORT=5555
      - NUM_DICE=3
      - NUM_GAMES=5
      - STARTUP_DELAY=3
    depends_on:
      - bob

networks:
  game_network:
    driver: bridge
    name: dice_network
```

Listing 9: docker-compose.yml

Key configuration points:

- **Network**: Custom bridge network allows containers to communicate using hostnames
- **depends_on**: Ensures Bob's server starts before Alice's client
- **Environment variables**: Configure protocol parameters
  - `NUM_DICE`: Number of dice per player (default: 3)
  - `NUM_GAMES`: Number of games in match (default: 5)
  - `BOB_PORT`: Server port (default: 5555)
  - `STARTUP_DELAY`: Delay before Alice connects (default: 3s)
- **Port mapping**: Exposes Bob's port 5555 for debugging

6 Testing and Validation

6.1 Test Execution

The system was tested by building the Docker images and running a complete match of 5 games with 3 dice per player. Each game follows the complete four-phase protocol with full verification of cryptographic commitments.

6.2 Test Results

A test match of 5 games was executed with the following results:

Game	Alice Dice	Sum	Bob Dice	Sum	Winner
1	[6,5,5]	16	[4,6,3]	13	Alice
2	[6,4,5]	15	[2,5,3]	10	Alice
3	[5,1,5]	11	[2,5,5]	12	Bob
4	[4,2,5]	11	[1,4,4]	9	Alice
5	[4,6,5]	15	[2,6,1]	9	Alice
Match Result: Alice wins 4-1					

Table 3: Test match results (5 games with 3 dice)

Key observations:

- All commitment verifications succeeded (no cheating detected)
- Network communication worked correctly between containers
- Protocol execution completed within 1-2 seconds per game
- All four protocol phases executed correctly in each game
- Match tracking and final score calculation worked correctly
- Dice sums verified correctly (sum of individual dice values)

6.3 Sample Output Analysis

Game 1 output confirms: (1) Alice committed before knowing Bob's roll, (2) Bob rolled without knowing Alice's sum (hiding property), (3) Verification succeeded proving Alice didn't cheat (binding property), (4) Winner determined correctly (Alice: 16 vs Bob: 13).

7 Security Analysis

7.1 Threat Model

We assume:

- Both Alice and Bob follow the protocol (no arbitrary deviations)
- The network is insecure (adversary can observe all messages)
- Either Alice or Bob may attempt to cheat within protocol rules
- Cryptographic primitives (SHA-256, OS entropy) are secure

7.2 Attack Scenarios and Defenses

7.2.1 Attack 1: Alice Tries to Change Her Sum

Scenario: After seeing Bob's sum s_B, Alice wants to change her committed sum s_A to a winning sum s'_A.

Attack Strategy: Alice must provide (s'_A, d', r') such that:

- SHA-256$(s'_A||r') = C$ where C was her original commitment
- $\sum d'_i = s'_A$ (dice must sum to claimed value)

Defense: This requires finding a SHA-256 collision, which is computationally infeasible (security level: 2^{128} operations for collision resistance).

Probability of success: $\approx 2^{-128}$ (effectively impossible)

Additional defense: Bob verifies that $\sum d_i = s_A$, so Alice must also find valid dice values.

7.2.2 Attack 2: Bob Tries to Determine Alice's Sum

Scenario: Bob receives commitment C and wants to determine Alice's sum before rolling his dice.

Attack Strategy: Bob attempts dictionary attack over all possible sums.

For k=3 dice: Try sums $s \in \{3, 4, ..., 18\}$ (16 values)

Without nonce: Bob computes SHA-256(s) for each sum and compares with C - attack succeeds!

With nonce: Bob must try $16 \times 2^{256} \approx 1.8 \times 10^{78}$ combinations.

Defense: 256-bit nonce makes brute-force search infeasible. With computational power of 10^{18} hashes/second, attack would take approximately 10^{52} years.

Probability of success: Negligible (approximately 2^{-256} per guess)

7.2.3 Attack 3: Alice Reports Fake Dice

Scenario: Alice reports dice values that don't sum to her committed sum, or reports physically impossible dice values.

Defense 1 - Sum Verification: Bob verifies $\sum d_i = s_A$ before accepting the result.

Defense 2 - Dice Value Validation: Bob verifies each $d_i \in \{1, 2, 3, 4, 5, 6\}$.

Outcome: Bob detects cheating and aborts the game.

7.2.4 Attack 4: Replay Attack

Scenario: Adversary captures Alice's commitment from a previous game and replays it.

Defense: Each game uses a fresh random nonce generated by `secrets.token_hex()`, which uses OS entropy. The probability of nonce collision across games is 2^{-256}.

Probability of success: $\approx 2^{-256}$ (impossible in practice)

7.2.5 Attack 5: Bob Changes His Roll After Receiving Reveal

Scenario: Bob waits to see Alice's reveal before deciding what to send as his roll.

Why this fails: Bob must send his roll (Phase 2) before Alice reveals (Phase 3). The protocol enforces this ordering.

Defense: Sequential message exchange ensures Bob commits to his roll before learning Alice's result.

7.3 Security Properties Achieved

Property	Mechanism
Fairness	Commitment prevents both players from gaining advantage
Binding	SHA-256 collision resistance prevents Alice from changing result
Hiding	SHA-256 preimage resistance + 256-bit nonce prevents Bob from determining Alice's sum
Correctness	Deterministic winner calculation based on verified sums
Integrity	Dice sum verification ensures reported values match actual dice
Non-repudiation	All messages are transmitted and verified

Table 4: Security properties analysis

Note: The protocol does not provide:

- **Privacy**: Dice results are visible after reveal (intentional for game transparency)
- **Authentication**: No player identity verification (could be added with TLS certificates)
- **Denial of Service protection**: No mechanism to prevent Bob from refusing connections

In a production system, adding TLS encryption and mutual authentication would address some of these limitations.

7.4 Formal Security Argument

Theorem: Under the assumption that SHA-256 is collision-resistant and preimage-resistant, the protocol ensures that neither Alice nor Bob can gain an unfair advantage.

Proof sketch:

1. **Alice cannot cheat**: After sending $C = \text{SHA-256}(s_A||r)$, Alice is bound to (s_A, r) by collision resistance. To change her result, she must find (s'_A, r') such that $\text{SHA-256}(s'_A||r') = C$, which requires finding a SHA-256 collision. This requires $O(2^{128})$ operations (birthday attack), which is computationally infeasible.
2. **Bob cannot cheat**: Bob must choose his roll before learning s_A. Given C, determining s_A requires inverting SHA-256, which is preimage-resistant (requires $O(2^{256})$ operations). The dictionary attack over all possible sums is prevented by the 256-bit nonce, requiring $O(k \cdot 5 \cdot 2^{256})$ operations for k dice.

3. **Fairness**: Both players' results are determined before either learns the other's result. The protocol simulates "simultaneous" revelation despite sequential network communication. The expected outcome matches that of a fair simultaneous roll.

4. **Integrity**: Bob's verification of $\sum d_i = s_A$ ensures Alice cannot report false dice values or sums.

Therefore, the protocol achieves computational security based on the hardness assumptions of SHA-256, with security parameter $\lambda = 128$ bits (collision resistance).

8 Design Choices Summary

Choice	**Rationale**
SHA-256 hash function	NIST-approved, widely available, 128-bit collision resistance
256-bit nonce	Provides overwhelming security against brute-force and birthday attacks
TCP sockets	Reliable, ordered delivery suitable for turn-based protocol
JSON message format	Human-readable, easy to debug, sufficient for small messages
Newline delimiters	Simple parsing, clear message boundaries
Docker containers	Lightweight VM alternative, easy deployment, consistent environment
Python 3.11	Built-in crypto libraries, rapid development, clear code
3 dice default	Balanced gameplay (range 3-18), demonstrates protocol scalability
5 games per match	Sufficient to demonstrate match system, reasonable test duration
Sequential games	Simpler than parallel, sufficient for demonstration

Table 5: Design choices and rationale

9 Conclusions

This project successfully implemented a secure dice game protocol using cryptographic commitments. The implementation satisfies all requirements:

1. **Configurable Dice Game**: Players can roll k six-sided dice (configurable via environment variable)

2. **Fair Game Mechanics**: Winner determined by highest sum with proper verification

3. **Match System**: Multiple games tracked with final match winner determination

4. **Cheat Prevention**: SHA-256 commitment scheme ensures neither player can cheat

5. **Virtual Machines**: Docker containers provide isolated execution environments
6. **Complete Documentation**: All design choices and implementation details documented

Key achievements include:

- **Cryptographic Security**: Hash-based commitment with 256-bit nonce providing strong security guarantees (128-bit collision resistance, 256-bit preimage resistance)
- **Clean Protocol Design**: Four-phase commit-reveal protocol that is simple, verifiable, and extensible
- **Practical Implementation**: Real distributed system with Docker deployment demonstrating protocol in action
- **Comprehensive Testing**: Multiple test matches confirming correct operation and verification
- **Extensibility**: Protocol easily extends to different numbers of dice, different dice types (e.g., d20), or different comparison rules

This project demonstrates how commitment schemes enable fair protocols in distributed systems. The cryptographic nonces prevent dictionary attacks on low-entropy data, while proper dual verification (commitment + data integrity) ensures security. Docker containerization provides an effective platform for protocol simulation, and sequential protocol design achieves fairness equivalent to simultaneity through cryptographic binding.

The commit-reveal pattern has applications in online auctions, voting systems, blockchain smart contracts, and any distributed game requiring simultaneous moves without a trusted third party.