This report presents the design and implementation of a two-player Battleship game utilizing Paillier homomorphic encryption to ensure complete privacy of board positions. The system demonstrates privacy-preserving computation where a neutral game server can process player guesses and determine hits or misses without ever accessing unencrypted board data. This work bridges theoretical cryptography with practical application, showcasing how homomorphic encryption enables zero-trust architectures in multi-party computation scenarios.

Introduction

In modern networked gaming and distributed systems, trust is a fundamental concern. Traditional client-server architectures require players to trust the server with sensitive game state information. However, this trust model presents several vulnerabilities:

- Server compromise could expose all player data
- Malicious server operators could provide unfair advantages
- Players cannot verify server integrity without access to source code
- Centralized data storage creates single points of failure

Primary Objective: Design and implement a two-player Battleship game where:

1. Each player maintains a 10×10 game board with 5 ships
2. Ship positions remain cryptographically hidden from opponents and the game server
3. The game server can correctly process guesses and determine hits/misses
4. Players alternate turns until one player's fleet is completely destroyed

Security Requirements:

- Player boards must remain encrypted throughout gameplay
- The game server must process guesses without decryption
- No information leakage about ship positions beyond hit/miss results
- Each player maintains independent cryptographic keys

Approach

Paillier homomorphic encryption is in use, which supports additive homomorphism and scalar multiplication on encrypted data. This allows the game server to:

1. Retrieve encrypted board positions
2. Apply blinding factors using homomorphic multiplication
3. Return results to players for decryption
4. Determine hits/misses based on decrypted values (zero vs. non-zero)

Background

Homomorphic Encryption

Homomorphic encryption enables computation on encrypted data without decryption. Given an encryption function E and plaintexts m_1, m_2, a homomorphic encryption scheme allows:

Additive Homomorphism:

$E(m_1) \oplus E(m_2) = E(m_1 + m_2)$

Multiplicative Homomorphism:

$E(m_1) \otimes E(m_2) = E(m_1 \times m_2)$

Classification:

- **Partially Homomorphic Encryption (PHE):** Supports one operation (addition OR multiplication)
- **Somewhat Homomorphic Encryption (SHE):** Limited number of both operations
- **Fully Homomorphic Encryption (FHE):** Unlimited operations of both types

Paillier Cryptosystem

Introduced by Pascal Paillier in 1999, the Paillier cryptosystem is a probabilistic asymmetric algorithm that provides additive homomorphism.

Key Properties:

1. **Additive Homomorphism:** $E(m_1) \times E(m_2) = E(m_1 + m_2)$
2. **Scalar Multiplication:** E(m)^k = E(k × m)

3. **Semantic Security:** Computationally infeasible to gain information about plaintext from ciphertext

Mathematical Foundation:

Key Generation:

1. Select two large prime numbers p and q
2. Compute n = p × q
3. Compute λ = lcm(p-1, q-1)
4. Select generator g where gcd(L(g^λ mod n²), n) = 1
5. Public key: (n, g)
6. Private key: λ

Encryption:
$c = g^m \times r^n \bmod n^2$

where m is the plaintext and r is a random number in Z^*_n

Decryption:
$m = L(c^\lambda \bmod n^2) \times \mu \bmod n$

where $L(u) = (u-1)/n$ and $\mu = L(g^\lambda \bmod n^2)^{-1} \bmod n$

Homomorphic Properties:

$D(E(m_1) \times E(m_2) \bmod n^2) = m_1 + m_2 \bmod n$
$D(E(m)^k \bmod n^2) = k \times m \bmod n$

System Design

Component Responsibilities:

1. **Player Components:**
    - Maintain private board state (10×10 matrix)
    - Generate cryptographic keypairs
    - Encrypt board positions
    - Decrypt hit/miss results
    - Track opponent guesses
2. **Game Network:**
    - Store encrypted board data
    - Process player guesses using homomorphic operations
    - Apply blinding factors for security
    - Coordinate turn-based gameplay
    - Never access plaintext positions

Data Structures

Ship Class

```
class Ship:
  - name: string          # Ship identifier
  - size: integer         # Length (2-5 cells)
  - coordinates: list     # [(row, col), ...]
  - hits: set             # Track damaged positions
```

Ship Types and Sizes:

- Carrier: 5 cells
- Battleship: 4 cells
- Cruiser: 3 cells
- Submarine: 2 cells
- Destroyer: 2 cells

Player Class

```
class Player:
  - name: string
  - board: 10×10 matrix          # 0=water, 1=ship
  - guess_board: 10×10 matrix    # Track opponent guesses
  - ships: list[Ship]            # Fleet of 5 ships
  - encrypted_board: dict        # {(row,col): ciphertext}
```

- crypto: SimplifiedPaillier # Encryption engine
- hits_received: integer # Damage counter

SimplifiedPaillier Class

class SimplifiedPaillier:
- p, q: prime numbers # Private parameters
- n: integer # Public modulus (p×q)
- n_sq: integer # n²
- g: integer # Generator
- lambda_n: integer # Carmichael function

Game Flow

Phase 1: Setup

1. **Key Generation:**
    - Each player generates independent Paillier keypair
    - Public keys exchanged (private keys never shared)
2. **Board Initialization:**
    - Each player places 5 ships randomly
    - Ships cannot overlap or extend beyond board boundaries
    - Validation ensures proper placement
3. **Encryption:**
    - All 100 board positions encrypted
    - Encrypted values stored in game network
    - Original board remains private to player

Phase 2: Gameplay Loop

while no winner:
1. Current player views their board and guess history
2. Current player submits guess (row, col)
3. Validation: check if position already guessed
4. Network retrieves encrypted value at (row, col)
5. Network applies blinding: E(value)^random
6. Defending player decrypts result
7. Determine outcome:
   - Decrypt to 0 → Miss
   - Decrypt to non-zero → Hit
8. Update game state

9. Check win condition (all ships sunk)
10. Switch players

Phase 3: Conclusion

- Winner announced
- Final boards revealed
- Statistics displayed (total turns, ships sunk)

Security Model

Threat Model:

Trusted:

- Each player's local device and private key
- Cryptographic primitives (encryption/decryption functions)

Untrusted:

- Game network server
- Network communication channels
- Opponent player

Security Guarantees:

1. **Confidentiality:** Ship positions remain encrypted except to the owning player
2. **Integrity:** Homomorphic operations preserve correctness of results
3. **Privacy:** Blinding prevents position inference from decrypted values
4. **Non-repudiation:** Each player controls their own keys

Attack Resistance:

- **Eavesdropping:** Encrypted communications protect against passive observation
- **Server Compromise:** Server cannot decrypt board positions
- **Collision Attacks:** Random ship placement prevents predictable patterns
- **Timing Attacks:** Constant-time operations in encryption/decryption

---

Implementation Details

Simplified Paillier Implementation

Our implementation uses small primes (p=61, q=53) for educational purposes. Production systems use 2048-bit or larger primes.

Key Generation

```
def __init__(self):
    self.p = 61
    self.q = 53
    self.n = self.p * self.q        # n = 3233
    self.n_sq = self.n * self.n     # n² = 10,452,289
    self.g = self.n + 1             # g = 3234
    self.lambda_n = (self.p - 1) * (self.q - 1)  # λ = 3120
```

Security Note: Small primes enable fast computation for demonstration but are cryptographically weak. Real implementations require:

- Prime numbers ≥ 1024 bits each
- Random prime generation
- Coprimality checks
- Generator validation

Encryption Function

```
def encrypt(self, plaintext):
    r = random.randint(1, self.n - 1)
    # c = g^m × r^n mod n²
    c = (pow(self.g, plaintext, self.n_sq) *
         pow(r, self.n, self.n_sq)) % self.n_sq
    return c
```

Properties:

- Probabilistic: Same plaintext produces different ciphertexts
- Random element r ensures semantic security
- Result space: [0, n²-1]

Decryption Function

```
def decrypt(self, ciphertext):
    # u = c^λ mod n²
    u = pow(ciphertext, self.lambda_n, self.n_sq)
```

```
    # L(u) = (u-1)/n
    l = (u - 1) // self.n
    # m = L(u) mod n
    return l % self.n
```

Correctness: Given c = g^m × r^n mod n², decryption recovers m through the Carmichael function and modular arithmetic properties of the construction.

Homomorphic Operations

```
def multiply_encrypted_by_constant(self, ciphertext, constant):
    # E(m)^k = E(m×k)
    return pow(ciphertext, constant, self.n_sq)

def add_encrypted(self, c1, c2):
    # E(m1) × E(m2) = E(m1+m2)
    return (c1 * c2) % self.n_sq
```

Usage in Battleship:

- Multiply by blinding factor: `E(cell)^r = E(cell×r)`
- If cell=0 (water): E(0×r) = E(0) → decrypts to 0
- If cell=1 (ship): E(1×r) = E(r) → decrypts to r (non-zero)

Board Management

Ship Placement Algorithm

```
def place_ship_random(self, ship):
    placed = False
    while not placed:
        # Choose orientation
        horizontal = random.choice([True, False])

        # Choose valid starting position
        if horizontal:
            row = random.randint(0, 9)
            col = random.randint(0, 10 - ship.size)
        else:
            row = random.randint(0, 10 - ship.size)
            col = random.randint(0, 9)

        # Validate placement
```

```
        coords = ship.place(row, col, horizontal)
        if all(self.board[r][c] == 0 for r, c in coords):
            # Place ship
            for r, c in coords:
                self.board[r][c] = 1
            placed = True
```

Validation Rules:

- Ships cannot overlap
- Ships must fit within board boundaries
- Rejection sampling ensures uniform random placement

Board Encryption

```
def encrypt_board(self):
    for row in range(10):
        for col in range(10):
            cell_value = self.board[row][col]  # 0 or 1
            encrypted_value = self.crypto.encrypt(cell_value)
            self.encrypted_board[(row, col)] = encrypted_value
```

Encryption Details:

- Each of 100 cells encrypted independently
- Encryption time: O(100) operations
- Storage: 100 ciphertext values per player
- Values remain encrypted until game conclusion

Homomorphic Hit Detection

This is the core cryptographic protocol enabling privacy-preserving gameplay.

```
def check_hit_homomorphic(self, guess_row, guess_col, opponent):
    # Step 1: Retrieve encrypted position
    encrypted_position = self.encrypted_board[(guess_row, guess_col)]

    # Step 2: Apply blinding factor
    blinding_factor = random.randint(2, 100)
    encrypted_result = self.crypto.multiply_encrypted_by_constant(
        encrypted_position, blinding_factor
    )

    # Step 3: Decrypt (only owner can do this)
```

```
    decrypted_val = self.crypto.decrypt(encrypted_result)

    # Step 4: Interpret result
    is_hit = (decrypted_val != 0)

    # Step 5: Update game state
    if is_hit:
        self.hits_received += 1
        for ship in self.ships:
            if (guess_row, guess_col) in ship.coordinates:
                ship.hits.add((guess_row, guess_col))
                if ship.is_sunk():
                    return "sunk", ship.name
                break
        return "hit", None
    else:
        return "miss", None
```

Protocol Analysis:

Information Flow:

1. Game network accesses E(board[row][col])
2. Network computes E(board[row][col])^r
3. Defending player receives E(board[row][col] × r)
4. Defending player decrypts to board[row][col] × r
5. Result interpretation: 0 = miss, non-zero = hit

Security Properties:

- **Network sees:** Encrypted values only (indistinguishable from random)
- **Attacker learns:** Only hit/miss outcome (minimal information)
- **Cannot infer:** Exact guess position (due to blinding)
- **Cannot forge:** Results (requires private key)

Why Blinding is Critical:

Without blinding:

Alice guesses (3,4) → Network retrieves E(value) → Bob decrypts to 0 or 1
Problem: Bob could cache E(0) and E(1), then match future guesses

With blinding:

Alice guesses (3,4) → Network retrieves E(value) → Applies E(value)^random
Bob decrypts to: 0 (miss) or random_number (hit)
Advantage: Every decryption produces different non-zero values

User Interface

Board Display

```
def display_own_board(self):
    print(f"\n{self.name}'s Board:")
    print("   " + " ".join(str(i) for i in range(10)))
    for i, row in enumerate(self.board):
        display = ["S" if cell == 1 else "~" for cell in row]
        print(f"{i}  " + " ".join(display))
```

Display Symbols:

- S: Ship position
- ~: Water
- X: Hit (on tracking board)
- 0: Miss (on tracking board)

Input Validation

```
def get_player_guess():
    while True:
        try:
            guess = input("Enter your guess (row col, e.g., '3 4'): ").strip()
            parts = guess.split()
            if len(parts) != 2:
                print("Please enter two numbers separated by a space.")
                continue
            row, col = int(parts[0]), int(parts[1])
            if 0 <= row <= 9 and 0 <= col <= 9:
                return row, col
            else:
                print("Coordinates must be between 0 and 9.")
        except ValueError:
            print("Invalid input. Please enter two numbers.")
```

Validation Checks:

- Correct format (two integers)
- Valid range (0-9 for both coordinates)
- Not previously guessed
- Error handling for malformed input

Security Analysis

Cryptographic Security

Paillier Security Assumptions

The security of our implementation relies on:

Decisional Composite Residuosity Assumption (DCRA): Given $n = p \times q$ and a random element $z \in Z^*_{n^2}$, it is computationally infeasible to determine whether z is an n-th residue modulo n^2.

Implications:

- Attacker cannot distinguish $E(m_1)$ from $E(m_2)$
- Ciphertexts appear uniformly random
- No partial information leakage from ciphertext observation

Semantic Security: The Paillier cryptosystem is IND-CPA (Indistinguishable under Chosen Plaintext Attack) secure under the DCRA assumption.

5.1.2 Key Size Analysis

Current Implementation:

- p = 61, q = 53 (6-bit primes)
- n = 3,233 (12-bit modulus)
- Security level: ~12 bits (trivially breakable)

Factorization Complexity: For our small n, trial division factors in microseconds.

Production Requirements:

Modulus Size	Security Level	Factorization Time (estimated)
1024 bits	~80 bits	Years (current technology)

2048 bits	~112 bits	Centuries
3072 bits	~128 bits	Infeasible

Recommendation: Minimum 2048-bit modulus for real applications.

Protocol Security

Information Leakage Analysis

What is revealed during gameplay:

1. Hit/miss outcomes (necessary for game functionality)
2. Number of turns taken
3. Timing of operations (potential side-channel)

What remains hidden:

1. Exact ship positions (until game conclusion)
2. Board layout of opponent
3. Future target positions
4. Ship orientation and clustering

Blinding Effectiveness:

Without Blinding:

Guess (3,4) → Decrypt E(board[3][4]) → Learn exact value (0 or 1)
Attack: Build mapping of positions to encrypted values
Result: Can predict future hits by ciphertext matching

With Blinding:

Guess (3,4) → Decrypt E(board[3][4] × r) → Learn 0 or random_value
Attack: Cannot build mapping (different random each time)
Result: Only binary hit/miss information revealed

Information Theory:

- Maximum information leaked per guess: 1 bit (hit or miss)
- Total information leaked: ≤ 100 bits (maximum guesses)
- Board entropy: $\log_2(C(100, 17)) \approx 74$ bits (ship placement combinations)

Attack Scenarios

Scenario 1: Passive Network Eavesdropping

- **Attack:** Observe all encrypted communications
- **Defense:** Semantic security of Paillier encryption
- **Outcome:** Attacker learns nothing beyond public information

Scenario 2: Compromised Game Server

- **Attack:** Server attempts to decrypt board positions
- **Defense:** Private keys never shared with server
- **Outcome:** Server cannot access plaintext, game remains secure

Scenario 3: Malicious Player

- **Attack:** Player lies about hit/miss results
- **Defense:** None in current implementation
- **Limitation:** Requires trust in honest decryption
- **Mitigation:** Could add zero-knowledge proofs (future work)

Scenario 4: Cryptanalysis Attempts

- **Attack:** Factor n to recover private key
- **Defense:** Computational hardness of factorization
- **Outcome:** Infeasible with proper key sizes (2048+ bits)

Scenario 5: Side-Channel Attacks

- **Attack:** Timing analysis of encryption/decryption
- **Defense:** Constant-time operations (not implemented)
- **Risk:** Low for this application, but present

Comparison with Traditional Implementation

Aspect	Traditional Battleship	Homomorphic Battleship
Server Trust	Required	Not required
Data Exposure	All boards visible to server	Encrypted data only
Cheating Risk	High (server can assist players)	Low (server cannot see boards)
Computation	Simple comparisons	Homomorphic operations

Performance	Fast (< 1ms per check)	Moderate (~10-50ms per check)
Complexity	Low	High
Privacy	None	Strong

Limitations and Vulnerabilities

Current Vulnerabilities:

1. **Weak Parameters:**
    - 12-bit modulus easily factored
    - Educational only, not production-ready
2. **Honest Decryption Assumption:**
    - Players could lie about results
    - No cryptographic proof of correct decryption
3. **No Forward Secrecy:**
    - If private key compromised, entire game history exposed
    - Could implement key rotation
4. **Replay Attacks:**
    - Encrypted values remain constant
    - Could implement nonce-based encryption
5. **Side-Channel Leakage:**
    - Timing variations in operations
    - Power consumption patterns

Mitigation Strategies:

1. Use Production Library:

```
 from phe import paillier
public_key, private_key = paillier.generate_paillier_keypair(n_length=2048)
```

2. Add Zero-Knowledge Proofs:

- Prove decryption correctness without revealing key
- Use zk-SNARKs or Bulletproofs

3. Implement Verifiable Computation:

- Require cryptographic commitments to results
- Enable third-party verification

4. Constant-Time Operations:

- Eliminate timing side-channels
- Use constant-time modular exponentiation

Outcomes:

1. Setup Phase Output

```
============================================================
  HOMOMORPHIC BATTLESHIP - Full Two-Player Game
============================================================

Rules:
- Each player has a 10x10 board
- Ships: Carrier(5), Battleship(4), Cruiser(3), Submarine(2), Destroyer(2)
- Players take turns guessing coordinates
- First to sink all opponent's ships wins!
- Boards are encrypted - even the game can't see ship locations!

============================================================
SETUP PHASE
============================================================

Generating encryption keys...

[Alice] Setting up board...
[Alice] All ships placed!

Alice's Board:
   0 1 2 3 4 5 6 7 8 9
0  ~ ~ S S S S S ~ ~ ~
1  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
2  S ~ ~ ~ ~ ~ ~ ~ ~ ~
3  S ~ ~ ~ ~ S S S ~ ~
4  S ~ ~ ~ ~ ~ ~ ~ ~ ~
5  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
6  ~ ~ ~ ~ S S ~ ~ ~ ~
7  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8  ~ ~ S S ~ ~ ~ ~ ~ ~
9  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

[Bob] Setting up board...
[Bob] All ships placed!

Bob's Board:
   0 1 2 3 4 5 6 7 8 9
0  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
1  ~ ~ S S S S S ~ ~ ~
2  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
3  ~ S ~ ~ ~ ~ ~ ~ ~ ~
4  ~ S ~ ~ S S S ~ ~ ~
5  ~ S ~ ~ ~ ~ ~ ~ ~ ~
```

```
6  ~ S ~ ~ ~ ~ ~ ~ ~ ~
7  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8  ~ ~ ~ S S ~ ~ ~ ~ ~
9  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

[Alice] Encrypting board positions...
[Bob] Encrypting board positions...

============================================================
GAME START!
============================================================
```

2. Turn 1 - Alice's Turn:

```
============================================================
TURN 1 - Alice's Turn
============================================================

Alice's Board:
   0 1 2 3 4 5 6 7 8 9
0  ~ ~ S S S S S ~ ~ ~
[... board display ...]

Alice's Tracking Board (Your Guesses):
   0 1 2 3 4 5 6 7 8 9
0
[... empty board ...]

Alice, make your guess!
Enter your guess (row col, e.g., '3 4'): 1 4

[Network] Processing encrypted guess at (1, 4)...
[Network] Computing with encrypted data (opponent's board stays hidden)...

*** HIT! ***
Bob's ship was hit at (1, 4)!

Press Enter to continue to next turn...
```

3. Turn 5 - Bob Sinks a Ship:

```
============================================================
TURN 5 - Bob's Turn
============================================================

Bob, make your guess!
Enter your guess (row col, e.g., '3 4'): 2 0
```

[Network] Processing encrypted guess at (2, 0)...
[Network] Computing with encrypted data (opponent's board stays hidden)...

*** HIT! ***
*** You sunk Alice's Cruiser! ***

Press Enter to continue to next turn...

4. Final Turn - Game Conclusion:

[Network] Processing encrypted guess at (8, 2)...
[Network] Computing with encrypted data (opponent's board stays hidden)...

*** HIT! ***
*** You sunk Bob's Destroyer! ***

==
GAME OVER!
==

🎉 Alice WINS! 🎉
All of Bob's ships have been sunk!
Total turns: 47

Alice's Final Board:
  0 1 2 3 4 5 6 7 8 9
[... final positions revealed ...]

Bob's Final Board:
  0 1 2 3 4 5 6 7 8 9
[... final positions revealed ...]