Midnight Matching Pennies

Privacy-Preserving Game Implementation with Compact Smart Contracts

Ramiro Seltzer

July 27, 2025

Abstract

The following document presents a step-by-step approach to the Matching Pennies game based on Compact Smart Contract programming language. Throughout our document, we explore blockchain, zero-knowledge proofs, as well as privacy maintaining strategies such as commitment schemes to provide fairness on decision-taking games.

Contents

1 Introduction

The Matching Pennies game represents a classic example of a basic game where two players simultaneously choose one side of a coin. While trivial to implement in real life, blockchain implementation presents unique challenges due to the inherent transparency of blockchain ledgers.

This implementation leverages the Midnight Network's privacy-preserving capabilities and the Compact Smart Contract language to solve the fundamental problem of simultaneous commitment in a public blockchain environment.

2 Development Environment Setup

2.1 Overview

To implement our Matching Pennies solution, we must first establish a proper development environment following the Midnight Network Developer Tutorial. The setup process involves several critical components:

- Lace Wallet configuration and token acquisition
- Zero-Knowledge proof server initialization
- Compact compiler installation and configuration
- Development environment validation

2.2 Lace Wallet & Token Acquisition

The first step involves setting up a local Lace Wallet instance. The Midnight Network provides an interface for developers to acquire free test tokens (tDUST) necessary for development operations.

Test Credentials:

```
User: TestWallet
Password: TestWallet1234
Address: addr_test1kjwksfp8x2tachehsfvufsdl35ljg5cxzdcysjdn6ntadspyxn3q...
```

2.3 Zero-Knowledge Proof Server

The ZK server enables users to validate, secure, and shield transactions within the network. By simulating a real-life network locally, we must initialize the ZK server before performing any test operations.

Listing 1: Docker ZK Server Initialization

```
1 docker run -p 6300:6300 midnightnetwork/proof-server -- \
2   'midnight-proof-server --network testnet'
```

This configuration exposes the container's default port 6300, providing an interface for network communication and transaction validation.

2.4 Environment Validation

Following the tutorial guidelines, we perform a test transaction to validate our setup. The transaction involves sending 1 tDUST to a test address, confirming that our ZK server successfully processes and handles the request.

3 Example Contract Compilation

Before implementing our Matching Pennies game, we verify that our local network correctly handles contracts between wallets using the `example-counter` repository.

3.1 CLI Wallet Generation

Example Wallet Configuration:

```
Seed: 30c112855c5942f935e821d1d882f4bb4c436876caca1115f8935e77abb17e8b
Address: mn_shield-
    addr_test1sj5j3p2ajkar962ek8tyu87nwvwlmzlmpga8l6my7nrf9cn8s4eq...
```

3.2 Transaction Flow Validation

The validation process follows a structured approach:

1. **Transaction Initiation:** Sending test DUST tokens from Lace Wallet to the generated CLI wallet
2. **ZK Proof Generation:** The zero-knowledge proof server processes the transaction privately
3. **Blockchain Settlement:** The transaction is recorded on the Midnight network
4. **Local Wallet Update:** The CLI wallet receives and confirms the transaction

This successful transaction confirms that our connection between the local wallet and Lace wallet has been established correctly, with our ZK server functioning as the verification intermediary.

4 Matching Pennies Implementation

4.1 Objective

We want two players (A & B) to pick one side of a coin. In real life, this task is straightforward—both players pick a side and simultaneously reveal their choices to determine the winner. No party can interfere with the other player's decision, and no player can know the other's decision beforehand.

However, based on the nature of blockchains, all ledger information values remain public. Due to this constraint, we want to preserve the privacy of each player's decision before actually revealing their choice.

4.2 Game Implementation

4.2.1 Game joining (`joinGame`)

To play Matching Pennies, exactly two players must join the game. If a third player attempts to connect, the smart contract handles the logic to forbid their participation.

4.2.2 Commitment Scheme (`commitCoin`)

Once two players have joined, our structural approach implements a cryptographic commitment scheme. Both players must make a decision on a coin side, commit to it, and cannot change their decision afterward.

This commitment ensures players have decided on a coin side *a priori* while providing no insight into the actual value.

Security Implementation: Our implementation leverages Compact's built-in utilities and functions. Specifically, we commit a hashed value based on the coin side and salt value, which provides randomness to the hash. Compact provides the `persistentCommit<T>` function specifically designed to securely commit values.

We enforce the commitment policy with ledger boolean values `player1_commitment` and `player2_commitment` that verify whether a player has previously committed. No player can commit more than once per match.

4.2.3 Reveal Phase (`revealCoin`)

Once both players have committed to their coin choices, they proceed to the reveal phase. Each player submits their chosen coin side (heads or tails) along with the salt used in their commitment.

The `revealCoin` circuit verifies that:

- The lobby is full and no external players can reveal a coin, except for `player1` and `player2`
- The player hasn't previously revealed their choice
- The stored committed value matches the coin and salt produced commitment

Due to Compact's privacy features on circuit parameters, players can safely call `revealCoin` with their secret values without external observers accessing the data.

4.2.4 Winner Determination (`playMatchingPennies`)

The `playMatchingPennies` circuit determines the winner based on the game's rules, ensuring fair outcomes by enforcing:

- All players must be connected
- All players must commit before revealing
- All players must reveal before playing
- No player can reveal before the other commits

Winner Logic:

Listing 2: Winner Determination Logic

```
if(player1_coin_choice == player2_coin_choice){
    winner = player1;  // Matching choices
} else {
    winner = player2;  // Different choices
}
```

To allow subsequent matches, we reset the ledger's default values after each game completion.

4.3 Code Compilation

We verify successful compilation by running the compiler on the `MatchingPennies.compact` file:

Listing 3: Compilation Commands

```
# Check compiler version
compactc --version  # Expected: Compactc version: 0.24.0

# Compile the contract
compactc.bin MatchingPennies.compact .
```

Expected Project Structure:

```
MidnightMP/
          compiler/
          contract/
          keys/
          MatchingPennies.compact
          node_modules/
          package.json
          package-lock.json
          scripts/
          test/
          zkir/
```

Listing 4: Successful compilation

```
# Expected output after compiling
Compiling 4 circuits:
  circuit "commitCoin" (k=14, rows=10282)
  circuit "joinGame" (k=10, rows=277)
  circuit "playMatchingPennies" (k=10, rows=428)
  circuit "revealCoin" (k=14, rows=10247)
Overall progress [====================] 4/4
```

4.4 Challenges

Throughout the implementation of Matching Pennies, a set of challenges emerged due to the recentness & the lack of practical examples implemented on Compact's language.
For instance, Compact's documentation provides a bounded set of examples and restrictions when it comes to testing implementations. In order to approach these challenges, tools like Midnight's Discord server, community forums & their website's interactive AI search helped to make my way through the documentation easier.
A good approach to build Compact software given the state-of-the-art is to implement small code & leverage the AI trained on documentation bot on Discord to correct type, functions or syntax errors.

5 Testing & Validation

We provide a test suite to verify the smart contract's functionality. However, given Compact's recent development and closed-source nature, testing presents challenges that require additional time investment.

A future development area for Midnight & Compact involves implementing a plug-and-play testing framework locally, allowing developers to avoid uploading contracts to the blockchain and using the Testnet, thereby reducing development time and improving the overall development experience.

6 Conclusion

This project, built without prior blockchain experience, demonstrates the ability to quickly learn a new programming language, adapt to its unique features, and deliver a working solution under time constraints.

The implementation successfully addresses the core challenge of maintaining privacy and simultaneity in blockchain-based games through:

- Cryptographic commitment schemes
- Zero-knowledge proof integration
- Decentralized game logic enforcement

Additionally, this project has opened new avenues for exploring decentralized networks and blockchain technologies, enriching developer's toolkit with privacy-preserving smart contract development capabilities.

6.1 Future Improvements

Several enhancement opportunities emerged during development:

- **Privacy Enhancement:** Minimizing exposure by keeping player addresses private
- **Scalability:** Adding a queue system for players waiting to join when two players are already engaged
- **Analytics:** Storing player history with privacy-preserving structures to analyze patterns while maintaining anonymity
- **Testing Framework:** Developing comprehensive local testing capabilities