Mini Casino DApp

Blockchain-Based Blackjack and Token Casino with Gamification

Kwong Chi Yan
COMP4541 - Spring 2025

May 14, 2025

1 Project Summary

This project presents a decentralized application (DApp) that functions as a mini casino platform. It features an on-chain, playable game of Blackjack and Slots Machine supported by an ERC-20 token (CTKN) that players use as casino chips. Players can acquire CTKN by swapping ETH and can later sell them back to the contract.

2 Ethereum Network and Deployment

- **Testnet Used:** Sepolia Ethereum Testnet
- **Live Demo:** https://kacok007.github.io/casino-dapp/
- **Source Code:** GitHub Repository

3 Smart Contract Overview

3.1 Smart Contract Logic

- `CasinoToken` contract handles token minting, buying, and selling
- `CasinoGame` contract manages game logic, including:
  - Blackjack game state and dealer logic
  - Slots game outcomes and payouts
- Both contracts are designed to be secure and efficient

3.2 CasinoToken (ERC-20)

- Custom token: `CTKN`
- Exchange rate: 1 ETH = 10,000,000 CTKN
- Users can buy/sell CTKN using ETH
- Owner can withdraw ETH and mint new tokens

3.3 CasinoGame

- Unified contract supporting:
  - Blackjack with actual playable logic and deck tracking
  - Slots Machine with random number generation
- Admin functions:
  - Withdraw profit
  - Update house edge
  - Set game timeout
- Reserve management for CTKN to ensure game payouts
- Pseudo-randomness using `block.timestamp` and `block.difficulty`

- Game state management for Blackjack and Slots

4 Functionality Demonstration

4.1 Blackjack Gameplay

- Users start a game with CTKN as the bet
- The deck is shuffled and cards drawn without replacement
- Users interact via `hit()` and `stand()` functions
- Dealer logic follows standard Blackjack rules
- Outcomes include win, loss, push, or bust
- Winnings are automatically transferred to the user's balance
- Game state is reset after each round

4.2 Slots Machine

- Users can play Slots with CTKN as the bet
- Random number generation determines slot outcomes
- Payouts are based the randomly generated numbers:
  - Jackpot: 10x payout
  - Small Win: 2x payout
  - No match: 0 payout
- Game state is reset after each round
- Winnings are automatically transferred to the user's balance

4.3 Token Swap System

- Users can buy CTKN tokens by sending ETH
- Tokens are transferred directly from the contract's balance
- Selling CTKN transfers ETH back to the user
- Exchange rate is safely handled using wei units to avoid precision issues
- Users can check their CTKN balance via the contract

5 Frontend Features

- React + Vite web interface
- MetaMask wallet integration
- Live CTKN balance, swap UI, and interactive Blackjack game
- Deployed to GitHub Pages

6 User Interface

Gamepage

The game page allows users to select between Blackjack and Slots. The interface displays the current CTKN balance, the option to buy/sell tokens, and the game controls.

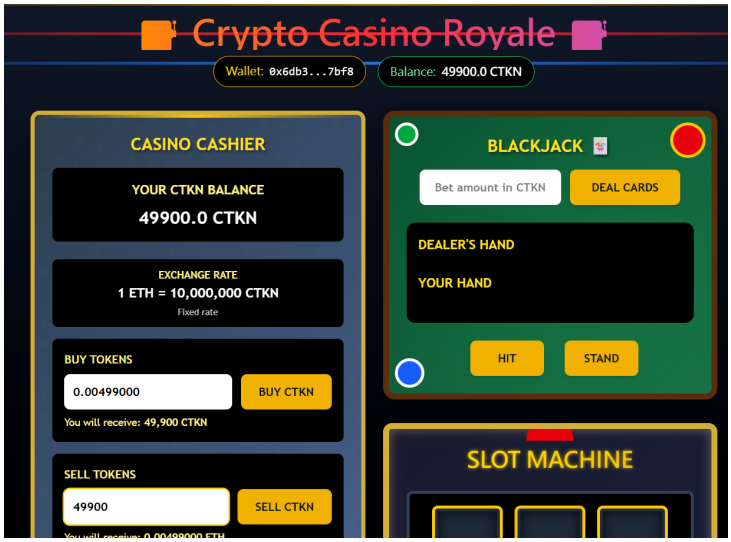

Figure 1: Game Page with Blackjack and Slots options

TokenSwap

The token swap interface allows users to buy CTKN with ETH or sell CTKN back to the contract or approve CTKN to CasinoGame contract. The current exchange rate is displayed, and users can input the amount they wish to swap. The interface also shows the expected amount of CTKN or ETH they will receive.

Blackjack Game

The Blackjack game interface displays the player's hand, dealer's hand, and the current bet amount. Users can choose to hit or stand, and the game will automatically resolve the outcome based on standard Blackjack rules. The interface also announces the result of the game (win, loss, push, or bust) and updates the CTKN balance accordingly.

Slots Game

The Slots game interface allows users to place a bet and spin the slots. The outcome is determined by random number generation, and the interface displays the result of the spin along with any winnings. The interface also announces the result of the game (jackpot, small win, or no match) and updates the CTKN balance accordingly.

7 Testing and Logs

Automated Testing

For testing, I used the the platform provided by the course, https://mycontract.fun/ to auto-generate test cases and testing logs. The CasinoToken was tested for the following 13 scenarios:

Buying Tokens

- `testBuyTokens`: Verifies that calling `buy()` with a nonzero ETH value gives the expected number of tokens.
- `testReceiveFallbackBuy`: Tests that sending ETH via the fallback (`receive`) automatically calls `buy()`.
- `testBuyNoETH`: Ensures that calling `buy()` with 0 ETH reverts with the expected error message.

Selling Tokens

- `testSellTokens`: Tests a successful sell operation (buy tokens first, then sell a portion).

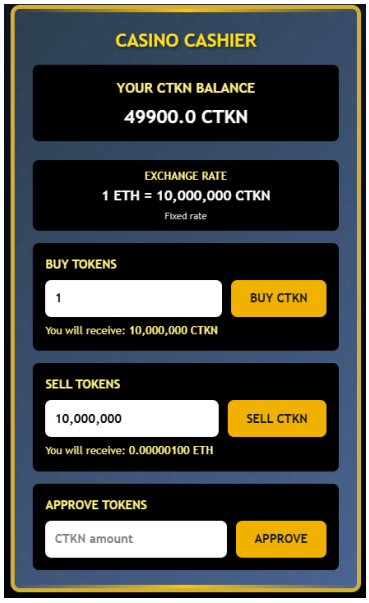

Figure 2: Token Swap Interface

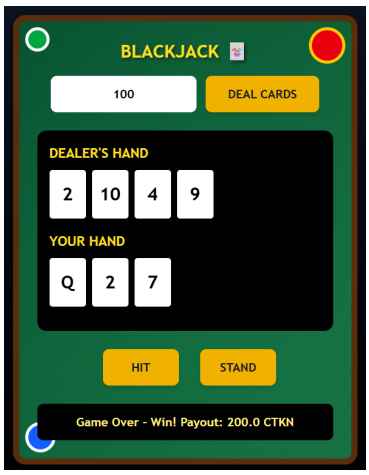

Figure 3: Blackjack Game Interface

- `testSellWithoutTokens`: Verifies that attempting to sell more tokens than owned reverts.
- `testSellNotEnoughETHReserve`: Checks that selling fails when the contract lacks sufficient ETH reserve.

Administrative Functions

- `testMintByOwner`: Tests that the owner can mint tokens to the contract's reserve.
- `testMintByNonOwner`: Ensures a non-owner cannot mint tokens.
- `testSetRateByOwner`: Verifies that the owner can update the token exchange rate.
- `testSetRateByNonOwner`: Checks that a non-owner cannot update the rate.
- `testSetRateZero`: Tests that setting the rate to zero reverts.
- `testWithdrawETHByOwner`: Confirms the owner can withdraw ETH from the contract.
- `testWithdrawETHByNonOwner`: Ensures a non-owner cannot withdraw ETH.

Figure 4: Slots Game Interface

Fallback/Receive Handling

- Covered by `testReceiveFallbackBuy` (no additional explicit tests needed).

The CasinoToken passed all tests successfully, confirming the expected behavior of the contract. The CasinoGame was tested for the following 12 scenarios:

Slots Game Tests

- `testPlaySlotsInsufficientAllowance`: Verifies that `playSlots()` fails if the player's token allowance is insufficient.
- `testPlaySlotsInsufficientFunds`: Ensures `playSlots()` reverts when the contract lacks sufficient funds.
- `testPlaySlotsNormalFlow`: Tests the normal execution flow of `playSlots()` (checks state updates).

Blackjack Game Tests

- `testStartBlackjack`: Validates proper initialization of a blackjack game.
- `testBlackjackTimeoutByHit`: Tests automatic game timeout via `hit()` after exceeding time limit.
- `testBlackjackStandFlow`: Verifies game completion via `stand()` action.
- `testOwnerTimeoutGame`: Checks admin-forced timeout functionality.

Admin Function Tests

- `testWithdrawProfitAccessControl`: Confirms only owner can call `withdrawProfit()`.
- `testWithdrawProfit`: Tests proper profit withdrawal by owner.
- `testUpdateHouseEdgeAccessControl`: Ensures only owner can update house edge.
- `testUpdateHouseEdge`: Validates house edge parameter update by owner.

Security Tests

- `test_arbitraryCallerCannotUpdateOwnerOnly`: Verifies critical owner-only functions reject non-owner calls.

The CasinoGame passed 9 out of 12 tests, with 3 tests failing due to wrong setup of the auto generated test cases. The failed tests were:

- `testBlackjackTimeoutByHit`: [FAIL: revert: No active game or timed out]
- `testPlaySlotsInsufficientFunds`: [FAIL: revert: Insufficient balance]

- `testWithdrawProfit`: [FAIL: revert: No active game or timed out]

These failures are due to the wrong setup of the auto generated test cases. The testBlackjackTimeoutByHit and testWithdrawProfit are false positive, as the logs showed that the output follow the expected behavior. The log showed the CasinoGame emitted the event and changed the state of the contract, therefore triggering the revert. That is the expected behavior of the contract. The testPlaySlotsInsufficientFunds is not completed, as the test case transferred CTKN from the CasinoToken contract to the CasinoGame contract, but the CasinoToken contract does not have enough CTKN to play the game.

The test cases and the logs can be found in the `test` folder of the GitHub repository.

Manual Testing

As the testPlaySlotsInsufficientFunds test case is not completed, I manually tested the CasinoGame contract by playing the game and checking the logs. The following table shows the actions taken and the results observed during manual testing:

Action	**Result**
Deploy CasinoToken contract	Contract deployed successfully
Deploy CasinoGame contract	Contract deployed successfully
Buy CTKN with 1 ETH	CTKN balance updated correctly
Not transfer CTKN to CasinoGame	CTKN balance remains 0
Approve CTKN to CasinoGame	CTKN allowance updated correctly
Play Slots with 1000 CTKN	Contract reverted: "Insufficient available funds"

The manual testing confirmed that the CasinoGame contract is working as expected.

8 Security Considerations

- **Reentrancy Attacks:** The contract uses the ReentrancyGuard to prevent reentrancy attacks, especially during token transfers and ETH withdrawals.
- **Access Control:** Only the contract owner can perform administrative functions, reducing the risk of unauthorized access.
- **Randomness:** The use of block properties for randomness with nonces to increase security. Future versions should integrate Chainlink VRF for further secure randomness, but this is not implemented in the current version.

9 Future Features

- Roulette integration
- Profit-sharing through CTKN staking
- On-chain achievements and NFT reward system
- Leaderboard and player performance metrics
- Chainlink VRF for secure randomness
- Reduce gas fees by optimizing contract code

Conclusion

This project showcases the integration of smart contracts, ERC-20 token economics, game logic, and a full-stack frontend to create an engaging, interactive decentralized casino DApp. With a strong foundation in place, future enhancements can bring richer gameplay and player incentives.