Course Name: Blockchain

Course ID: CDS528

Group Assignment: Blockchain DeFi Smart Contract Development

Project Title: European Call Option DeFi Platform

Group 11: Blockchain Geeker

Group Leader: ZHANG Weijian

Group Members: Liang Jianchong, Li Ya,LI Zixin, Qian Cheng

European Call Option DeFi Platform

A Blockchain-Based Decentralized Options Trading System

GitHub Repository: https://github.com/Blockchain-LU-Group/Blockchain-Group-Project
Demo Video: https://youtu.be/VU-VxNfOxWQ

1. Abstract

This project implements a decentralised finance (DeFi) application for European call options on ERC-20 tokens. The goal is to allow users to create, match, and exercise on-chain options without relying on a centralised exchange or broker. The core on-chain components are a factory contract that manages option instances and a EuropeanCallOption contract that defines the lifecycle of a single option, including premium payment, exercise, and expiry.

We use Solidity 0.8.19, OpenZeppelin libraries, and Hardhat for development, and build a Next.js + wagmi front-end that lets users connect wallets, create options, match open options as holders, and perform premium payment and exercise actions through a graphical dashboard. Testing combines unit and integration tests with Hardhat, mock ERC-20 tokens, and time manipulation utilities, while Slither is used for static security analysis. Deployment scripts target the Sepolia testnet and automate the deployment of factory, mock tokens, and example option contracts.

The system demonstrates how DeFi derivatives can be implemented with clear state machines, strong access control, and safety patterns such as reentrancy guards and explicit ERC-20 allowance checks. It also highlights current limitations (no oracle pricing, simple expiry model) and outlines future work such as integrating price feeds and supporting more complex option strategies.

2. Introduction

2.1 Background and Motivation

Options are fundamental derivatives in traditional finance, used for hedging, leverage, and structured products. However, access to options markets is often limited by jurisdiction, minimum account sizes, and centralised intermediaries. DeFi opens up the possibility of permissionless, transparent, and globally accessible options trading built entirely on smart contracts.

This project focuses on European call options on ERC-20 tokens. A European call gives the holder the right, but not the obligation, to buy an underlying asset at a fixed strike price after a certain expiration time. Bringing such instruments on-chain allows users to hedge price risk or take directional views using only a wallet and testnet tokens, without relying on a central exchange.

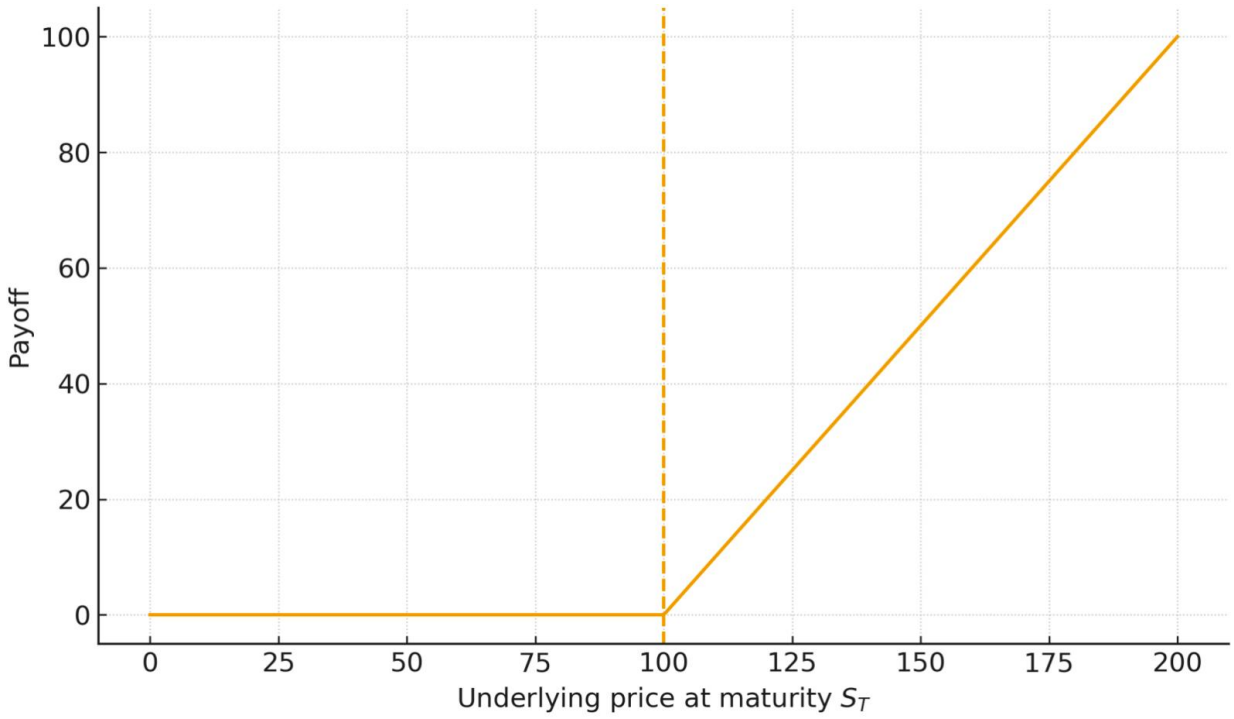

Figure 1. Payoff diagram of a European call option (Assume the strike price is 100)

2.2 Problem Statement

Traditional options trading platforms face several challenges:

- Requirement for trusted intermediaries and centralized exchanges
- Leak of sensitive information of institutional and high-net-worth traders
- High fees and barriers to entry
- Standardized Service and complex settlement waiting mechanism
- Limited accessibility and geographical restrictions

There is a need for a simple, educational on-chain options protocol that demonstrates the full lifecycle of a European call option—creation, matching, premium payment, exercise, and expiry— while remaining understandable and auditable for students and developers.

2.3 Target Users

- **Issuers** (sellers): Users who are willing to write call options, lock underlying tokens, and collect the premium and strike tokens if exercised
- **Holders** (buyers): Users who want the right to buy underlying tokens at a fixed price, paying a premium and potentially exercising profitably if the market price rises
- **Developers/learners**: Students who need a clean reference implementation of DeFi options with clear architecture and tests

2.4 Objectives

The primary objectives of this project are:

- Design a smart-contract-based European call option that encodes the option lifecycle as an explicit state machine
- Provide a factory contract to create and index multiple option instances
- Deliver a web front-end that allows non-technical users to interact with the contracts via a wallet
- Implement tests and basic security analysis to demonstrate good engineering practice

2.5 Scope, Assumptions and Constraints

Scope:

- Support European call options only (no puts, no American-style settlement mechanism)
- Use ERC-20 tokens as both underlying and strike assets
- Enforce a fixed exercise window: from expirationTime to expirationTime + 10 days

Assumptions:

- Underlying and strike tokens are standard ERC-20 with 18 decimals (or at least compatible with the call logic)
- Off-chain price discovery (e.g., from external markets) determines whether it is rational to exercise
- Users understand ERC-20 approvals and gas costs.

Constraints:

- Deployed only on Ethereum Sepolia and local Hardhat networks, not on mainnet (no real funds).
- No external oracles; the contract does not verify market price.
- Simplicity is preferred over full feature coverage of a production options protocol.

3. System Design and Architecture

3.1 Overview

The system is composed of:

- Smart Contracts Layer: Core business logic implemented in Solidity, including the EuropeanCallOption contract and OptionFactory contract
    - EuropeanCallOption contract
        - Encodes the lifecycle of a single option with states: Created → Active → Exercised or Expired
    - Manages roles: issuer (seller), holder (buyer), decentralized exchange(settlement platform) factory (creator contract)

    - Controls key operations: premium payment, exercise, expiry marking
    - OptionFactory contract
        - Creates new EuropeanCallOption instances
        - Stores an array of OptionInfo records, each tracking option address, issuer, holder, and creation time
        - Provides query functions for all options, matchable options, and options created by a given user
    - MockERC20 tokens
        - Simple ERC-20 implementation with configurable decimals and mint/burn functions, used for testing underlying and strike tokens
- Frontend Layer: Web-based user interface built with Next.js, React, and Web3 libraries (Wagmi, Viem)
    - Wallet connection and network status
    - Pages and components: option list, option creation page, option detail dashboard, and operation panel

3.2 Smart Contract Architecture

The smart contract system uses a factory pattern to manage option instances:

3.2.1 EuropeanCallOption Contract

This contract represents a single European call and uses ReentrancyGuard to protect state-changing external functions.

Core Parameters:

- underlyingAsset: Address of the ERC20 token serving as the underlying asset
- strikeAsset: Address of the ERC20 token used for strike price payment
- strikePrice: Price at which the option can be exercised (in strikeAsset units)
- expirationTime: Unix timestamp when the option expires
- contractSize: Amount of underlying asset covered by the option
- issuer: Address of the option seller who receives premium
- holder: Address of the option buyer who pays premium and can exercise
- status: Current state of the option (Created, Active, Expired, Exercised)

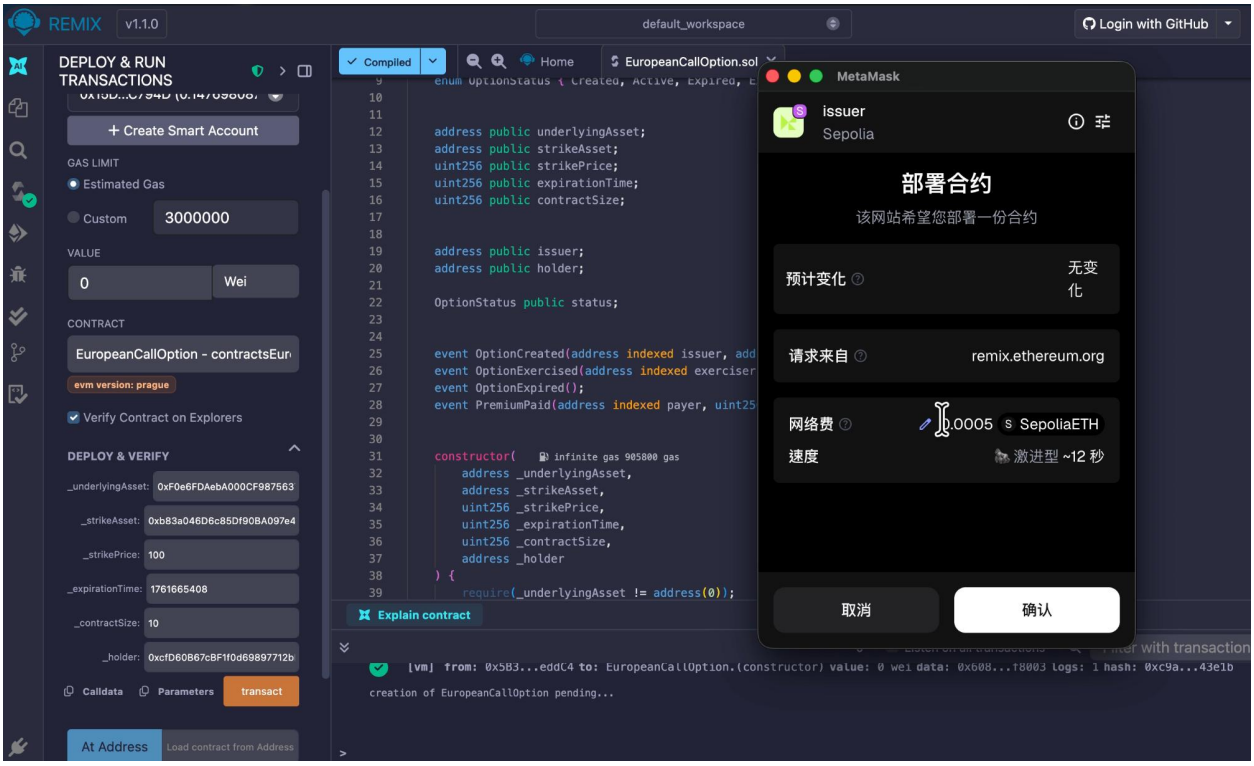

Figure 2. Core Parameters

Key Functions:

Constructor sets parameters and links the option back to the factory, initialising status = Created.

- payPremium(uint256 premium): only holder can call it, requires status = Created, Allows the holder to pay premium and activate the option
- exercised(): Enables the holder to exercise the option after expiration, transferring assets accordingly
- setHolder(address _holder): Factory-only function to assign a holder when matching occurs
- transferOption(address _newHolder): Allows holders to transfer their option rights
- isExercisable(): View function checking if the option can be exercised
- expireOption(): Public function to mark expired options

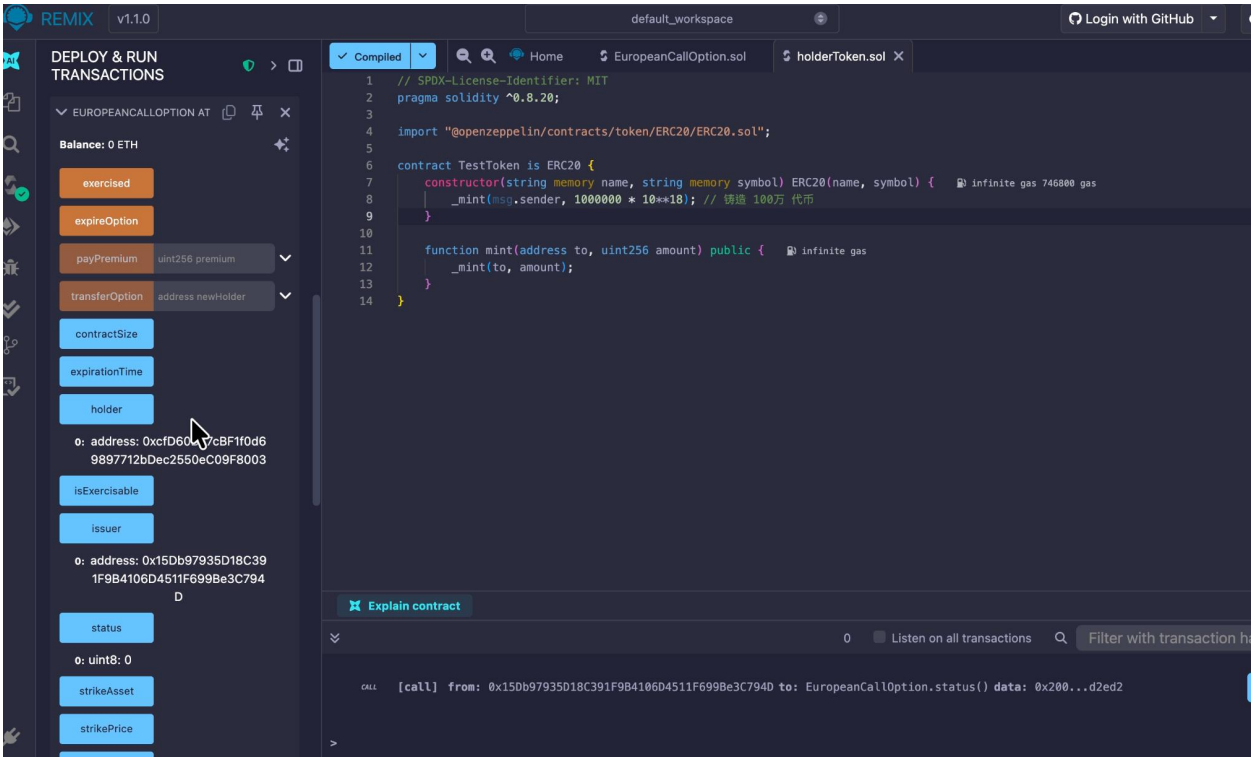

Figure 3. Key Functions

3.2.2 OptionFactory Contract

The OptionFactory contract serves as the central registry and creator for all option instances:

- createOption(...): Deploys a new EuropeanCallOption, records OptionInfo, and emits an event
- matchOption(address _optionAddress): Lets a user become holder of an un-matched option by calling setHolder on the option and updating OptionInfo
- getAllOptions(): Returns array of all option addresses
- getMatchableOptions(): Returns options available for matching
- getUserCreatedOptions(address _user): Retrieves options created by a specific user
- getOptionInfo(address _optionAddress): Returns detailed information about a specific option

3.3 Option Lifecycle

- Created: Option is created by an issuer through the factory, holder is set to zero address
- Active: Holder pays premium to issuer, option becomes active and ready for exercise
- Exercised: After expiration, holder exercises the option, paying strike price and receiving underlying asset
- Expired: Option passes the exercise window without being exercised

3.3 User Flows

Core flows:

- Option Creation (Issuer)
  - Issuer calls the factory to create an option with underlying asset, strike asset, strike price, expiration time, and contract size
  - OptionFactory.createOption deploys a new EuropeanCallOption with holder = address(0) and records it
- Option Matching (Holder)
  - Holder calls OptionFactory.matchOption(optionAddress) to become the holder of a matchable option
- Premium Payment (Holder)
  - Holder approves strike ERC-20 to the option, then calls payPremium(premium), which activates the option
- Option Exercise (Holder)
  - Within the exercise window, holder calls exercised(), paying strike tokens and receiving underlying tokens

4. Frontend dApp and User Experience

4.1 Technology Stack

- Next.js (App Router) and React for the front-end
- wagmi and viem for wallet connectivity and contract interactions
- Utility-class-based styling (e.g., Tailwind-style classes)

4.2 Main Screens and Components

- WalletConnect: connects MetaMask or other EVM wallets and shows network status
- OptionList: displays options from the factory with parameters, expiration, and status, and a "Match as Holder" button for matchable options
- OptionDashboard: shows details of a selected option, including parameters, roles, status, and exercise window
- OptionOperations: provides buttons for paying premium, exercising, and expiring options, and shows balances and allowances

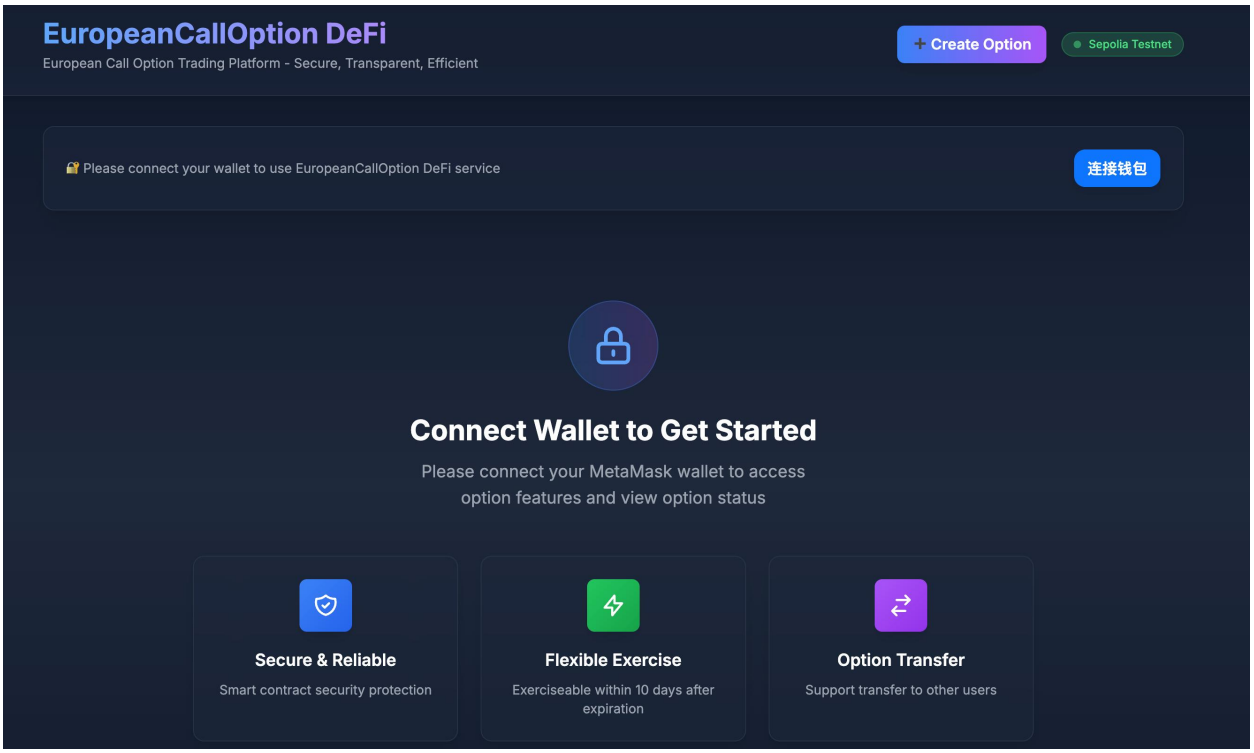

Figure 4. Main Screen

4.3 UX Considerations

- Validates user inputs such as addresses and non-negative values
- Detects network mismatches and warns the user
- Shows readable error messages for failed transactions and progress for pending ones
- Uses responsive card-based layout for a clear user experience

5. Testing and Quality Assurance

5.1 Testing Strategy

The project employs a comprehensive testing strategy covering unit tests, integration tests, and security analysis:

5.1.1 Unit Tests

Unit tests cover individual functions and edge cases:

- Constructor validation and initialization
- Premium payment flow with various scenarios
- Option exercise mechanics including timing constraints

- Holder setting and transfer functionality
- State transition validation
- Access control and permission checks
- Event emission verification
- Edge cases: zero addresses, zero values, invalid states

5.1.2 Integration Tests

Integration tests verify the complete option lifecycle:

- End-to-end flow: creation → matching → premium payment → exercise
- Asset transfer verification (premium, strike price, underlying asset)
- Balance calculations and state consistency
- Factory contract interactions with option instances
- Multiple options and concurrent operations

5.2 Security Analysis

Security analysis is performed using Slither static analyzer, which checks for:

- Reentrancy vulnerabilities
- Integer overflow/underflow
- Uninitialized state variables
- Unprotected external calls
- Access control issues
- Gas optimization opportunities

6. Deployment

6.1 Deployment Process

The deployment process follows best practices for blockchain applications:

- Environment Setup: Configuration of Sepolia testnet RPC endpoint and private keys
- Factory Deployment: Deploy OptionFactory contract to Sepolia testnet
- Mock Token Deployment: Deploy MockERC20 tokens for underlying and strike assets
- Option Deployment: Create option instances through the factory
- Contract Verification: Verify contracts on Etherscan for transparency and auditability
- Frontend Configuration: Update frontend with deployed contract addresses

6.2 Sepolia Deployment

Deployment scripts in the scripts directory:

- deploy_factory.js: deploys OptionFactory to Sepolia and saves addresses to deployments/sepolia_option.json.

- deploy_option.js: deploys factory (if needed), mock ERC-20 tokens, and a sample EuropeanCallOption, sets approvals, and updates deployment config
- verify_contract.js: verifies contracts on Etherscan using @nomicfoundation/hardhat-verify
- A new developer can configure .env, run the deploy script, and plug deployed addresses into the front-end config.

6.3 Deployment Information

Contracts deployed to Sepolia testnet:

- OptionFactory Address: 0x8000b7AE7a3C4138Dcc6DcE93E11051d1139c81D
- EuropeanCallOption Address: 0xF495Daa99e37fA6af97F47E324F90336aEc42B76
- Underlying Asset (MockERC20): 0x452f0232e083998d2f968b0602fb8208e7320df8
- Strike Asset (MockERC20): 0x424bd9e49e2049600deda9604d2a3ba836f99ee4

7. Results and Demonstration

The platform has been successfully implemented and deployed with the following achievements:

- Successfully deployed smart contracts to Sepolia testnet

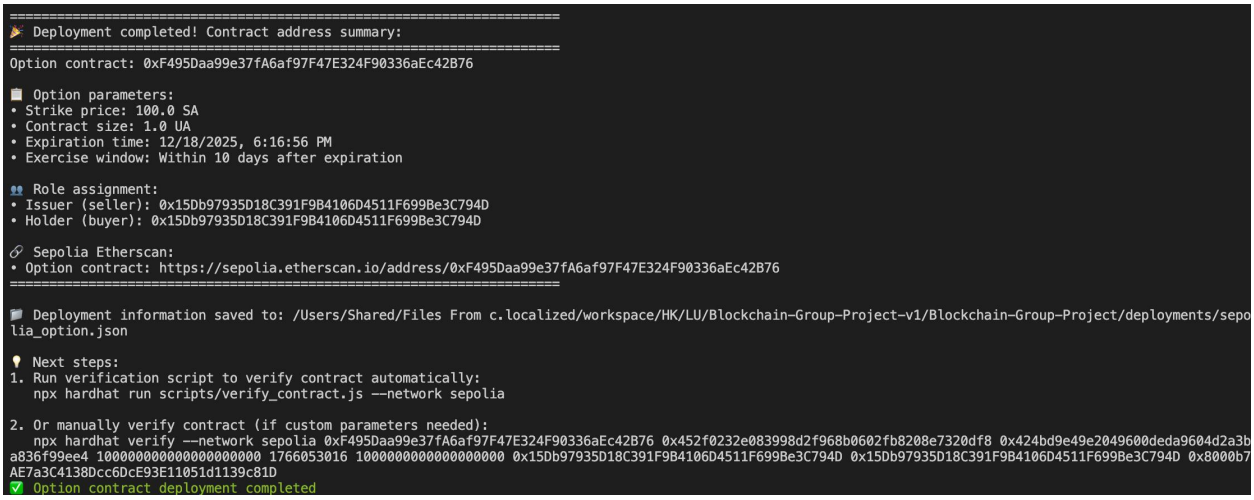

Figure 5. Deployment Summary

- Implemented comprehensive test suite with 100% coverage of core functions

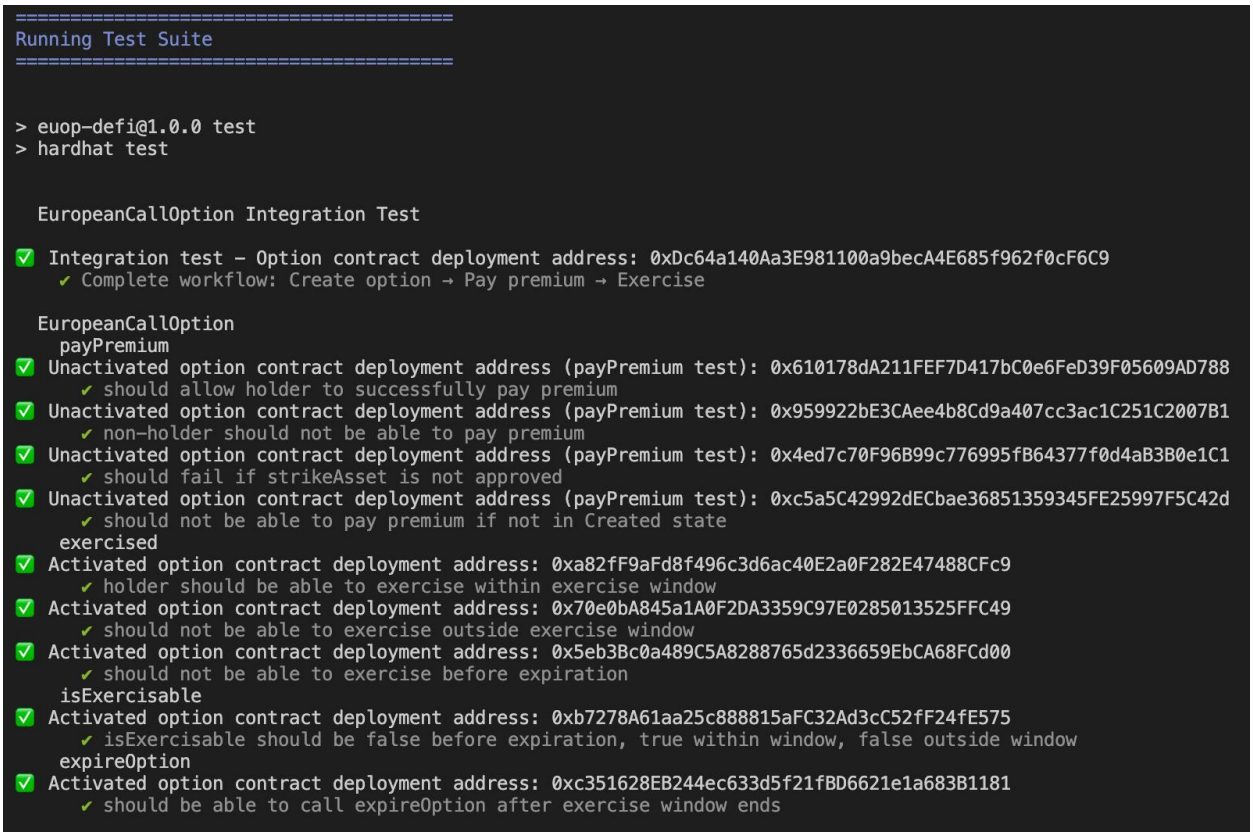

Figure 6. Test Summary

- Developed fully functional web interface with wallet integration
- Demonstrated complete option lifecycle: creation, matching, premium payment, and exercise
- Verified security through static analysis and extensive testing

7.1 User Interface

The frontend application provides an intuitive interface for all platform operations. Key features include real-time option listing, detailed option dashboards, transaction status tracking, and seamless wallet integration through RainbowKit.

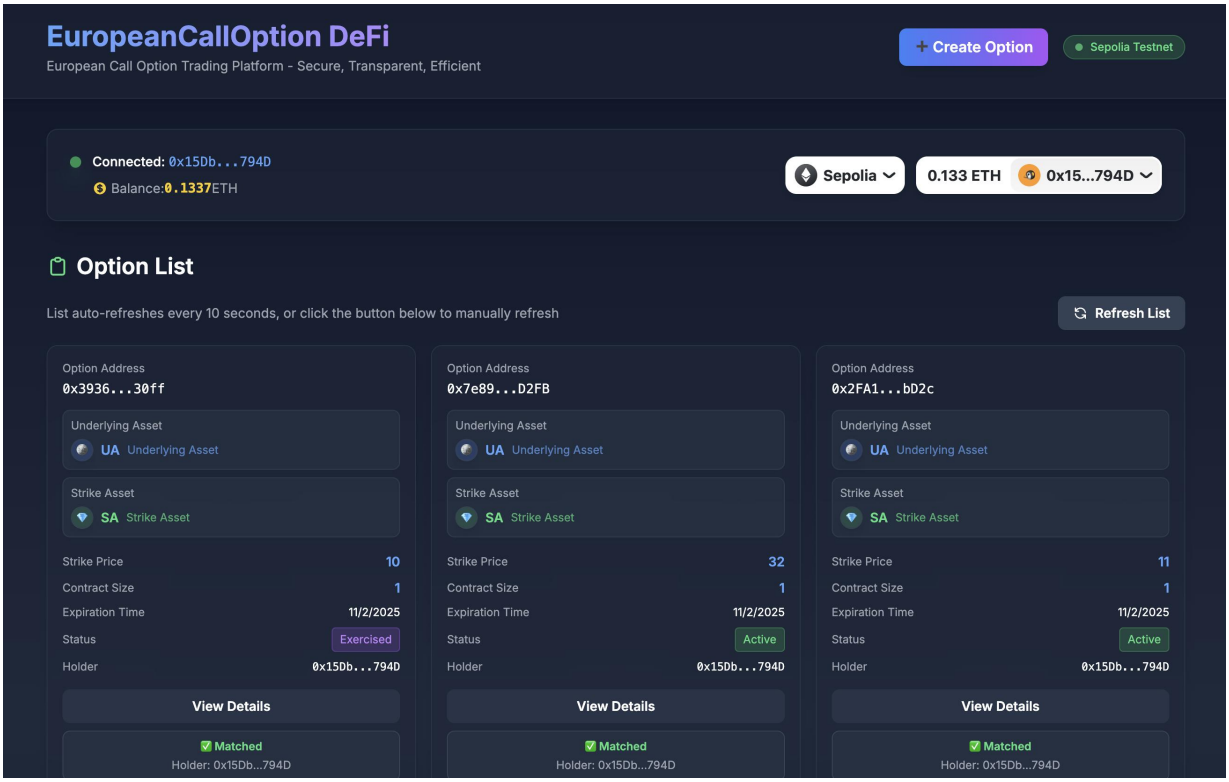

Figure 7. User Interface

8. Implementation Details

8.1 Technology Stack

Smart Contracts: Solidity 0.8.19, OpenZeppelin Contracts v4.9.0

Development Framework: Hardhat 2.17.0

Testing: Chai, Mocha, Hardhat Network Helpers

Frontend: Next.js 14, React 18, TypeScript 5.3

Web3 Libraries: Wagmi v2.0, Viem v2.0, RainbowKit

Styling: Tailwind CSS 3.4

Network: Sepolia Testnet (Ethereum)

Security Analysis: Slither static analyzer

8.2 Security Features

- ReentrancyGuard: All state-changing functions use OpenZeppelin's ReentrancyGuard to prevent reentrancy attacks

- Input Validation: All function parameters are validated for zero addresses, zero values, and logical constraints
- Access Control: Critical functions enforce proper authorization (only holder can pay premium/exercise, only factory can set holder)
- State Machine: Strict state transitions prevent invalid operations (e.g., cannot exercise before expiration)
- Safe Math: Solidity 0.8.19 provides built-in overflow/underflow protection
- Time Validation: Expiration and exercise window checks prevent premature or late operations

8.3 Frontend Implementation

The frontend application provides a comprehensive interface for interacting with the blockchain platform:

- WalletConnect: Integration with MetaMask and other Web3 wallets using RainbowKit
- Option List: Displays all available options with filtering capabilities
- Create Option: Form-based interface for creating new options with parameter validation
- Option Dashboard: Detailed view of individual options with status, parameters, and actions
- Option Operations: Interface for premium payment, exercise, and transfer operations
- Real-time Updates: Automatic refresh of contract state and transaction status
- Transaction Feedback: Clear success/error messages and loading states

9. Challenges and Solutions

State Management Complexity: Managing option states and transitions required careful design. Solution: Implemented clear state machine with enum types and strict transition rules.

Gas Optimization: Minimizing gas costs while maintaining functionality. Solution: Used optimizer with 200 runs, optimized storage patterns, and batch operations where possible.

Testing Time-Dependent Functions: Testing functions that depend on block timestamps. Solution: Used Hardhat Network Helpers to manipulate time in tests.

10. Future Enhancements

- Support for additional option types (put options), mechanisms (American-style settlement mechanism)
- Automated market making and order book functionality
- Option pricing models integrated with oracles (e.g., Chainlink)
- Secondary market for option trading and transfer
- Liquidation mechanisms for collateralized positions
- Multi-chain deployment and cross-chain interoperability

- Advanced analytics and portfolio management tools
- Mobile application development
- Integration with DeFi protocols (lending, yield farming)

11. Conclusion

This project successfully demonstrates the implementation of a decentralized European call option trading platform on the Ethereum blockchain. The system provides a secure, transparent, and automated solution for options trading without requiring traditional intermediaries.

Key accomplishments include: robust smart contract implementation with comprehensive security measures, user-friendly web interface, thorough testing, and successful deployment to a public testnet. The project showcases best practices in blockchain development, including proper access control, state management, event emission, and frontend integration.

The platform serves as a foundation for more complex DeFi derivative products and demonstrates the potential for blockchain technology to revolutionize traditional financial instruments.

12. References

12.1 Open Source Libraries

- OpenZeppelin Contracts v4.9.0: ERC20 interfaces, ReentrancyGuard (MIT License)
- Hardhat: Development environment and testing framework
- Wagmi: React Hooks for Ethereum
- Viem: TypeScript Ethereum client
- Next.js: React framework for production
- RainbowKit: Wallet connection UI components

12.2 Documentation

- Solidity Documentation: https://docs.soliditylang.org/
- Hardhat Documentation: https://hardhat.org/docs
- Ethereum Documentation: https://ethereum.org/en/developers/
- OpenZeppelin Contracts: https://docs.openzeppelin.com/contracts/