

CS438 Project Report

Spring 2025

Group Members:

Emre Koçer
Borhan Javadian
Neval Yaprak
Mert Karaçavuşoğlu

Overview:

The Project is a Decentralized Application (Dapp) that implements a simple Decentralized Finance (De-Fi) ecosystem in which users can perform basic financial operations such as swapping, borrowing and lending tokens.

These operations are implemented in Smart Contracts that can be deployed by users on demand.

Technical Details:

The Smart Contracts are written in Solidity and can be deployed by users on demand to be used for De-Fi functionalities. For testing and demonstration purposes we deploy the contracts on the Sepolia testnet.

The application consists of three modules that handle the required functionalities. These modules are implemented as separate Smart Contracts in individual Solidity files.

These modules are:

- AMM.sol : The Constant Product Market Maker Module
- DeFi.sol : The Lending Module
- Core.sol : The Core Module

Constant Product Automatic Market Maker Module:

AMM.sol contains the liquidity pool implementation that allows users to create a pool of two tokens, add and remove liquidity to the pool and swap tokens through the pool.

The tokens in the pool can be any ERC20 compliant token deployed on the blockchain.

The pool is implemented as a Constant Product Automatic Market Maker (AMM) where the ratio of the number of tokens in the pool is determined by a constant product.

This constant product is of the form $A \times B = K$

Where A is the first asset and B is the second asset.

AMM Functionalities:

1. Pool Creation:

Users can create a pool of two ERC20 compliant tokens and initialize it with a given liquidity.

2. Token Swapping:

Users can swap tokens by adding a single type of token to the pool and withdrawing a number of tokens of the other type. The number of tokens that can be swapped for each other is determined by the tokens value.

The tokens value in terms of each other is determined by the ratio of the tokens in the pool.

3. Liquidity Addition and Removal:

Users can add liquidity into existing pools from tokens in their possession. Users can both add liquidity by adding both types of tokens or only a single type of token. These users are called liquidity providers (LPs).

A fee of 0.3% is charged to users for each swap. These fees accumulate in the pool.

When a liquidity provider withdraws from the pool they also are rewarded by a share of the accumulated fees proportional to the amount of liquidity they contributed to the pool.

Lending Module:

DeFi.sol contains the implementation of the borrowing and lending functionalities.

This Smart Contract functions on top of a deployed pool and allows users to lend and borrow tokens of the types in the pool.

Lending Functionalities:

1. Borrowing:

Users can present one type of token as collateral and borrow tokens of the other type from the pool. These loans are then kept track of until they are repaid.

Since prices can fluctuate radically in a De-Fi ecosystem, the value of the collateral can decrease very quickly. We use the health factor of a loan to determine whether the collateral is sufficient for the loan amount.

2. Repaying and Liquidation:

Loans that are considered healthy can only be repaid by the user that initially took the loan. Whereas loans that are unhealthy can be paid for by liquidators who will then receive the collateral.

Liquidators can check the active loans and manually attempt to liquidate a loan. If the loan is unhealthy then the necessary amount of tokens to cover the loan will be transferred from the liquidator to the pool and the initial collateral will be transferred to the liquidator.

In either case of a loan being covered, the loan will get removed from the active list of loans.

3. Lending:

Users can also lend their tokens to the pool and then withdraw them later. As loans are taken and repaid by users, interest will be paid and accumulate in the pool.

When a lender withdraws from the pool they will also receive a reward from the total accumulated interest proportional to the amount that they had loaned.

Core Module:

Core.sol is used for the creation of new AMM pools. When users want to deploy a new pool, they pass the addresses of the tokens they wish to have in the pool and their initial liquidity to the Core.sol Smart Contract which then creates the pool by calling the necessary functions from the AMM.sol Smart Contract. The address of the deployed pool on the blockchain is then returned to the user.

The rest of the operations are then performed through the functions of the other Smart Contracts to which the address of the deployed pool is passed.

Logging Framework:

Logs of the state changes and operations performed by the Smart Contracts are kept in the Blockchain. The AMM and DeFi modules both have a set of events defined and these events are logged when appropriate as the different functions making up the application are called.

This allows keeping track of what is being performed by the users and since Blockchain provides data integrity, the integrity of the logs is also guaranteed.

Frontend User Interface:

The users can interact with the Smart Contracts through a simple HTML Frontend UI.

The users can specify the operations they would like to perform through the interface which will then be processed and performed by the smart contracts on the blockchain.

The results of the operation are then also displayed to the users through the Frontend UI.

Bonus Module: Arbitrage

The Arbitrage Module is an extra module that can be used by Arbitrageur Actors to perform Arbitrage between different deployed pools. This module was not demanded in the project but was implemented as a bonus.

This module first checks if there exists a valid arbitrage path between deployed pools by simulating swaps, and if valid paths exist, performs arbitrage through the most optimal path.

Code Explanation:

Below is an explanation of the various functions found in the different smart contract files of the project.

AMM.sol File:

1. Class Member Variables:

The AMM class keeps as member variables the address of the tokens it contains on the blockchain, the address of the AMM's owner, the liquidity of each token, and a few constant numeric values that are used during swap fee computations.

The class also keeps a mapping of user addresses to integers. This map is used to keep track of how many liquidity shares each liquidity provider has. The number of total liquidity shares is also kept track of.

2. Class Constructor:

The constructor takes the addresses of the tokens and the owners and sets them as the corresponding member variables.

3. Function initializeLiquidity:

This function initializes the liquidity of a pool when it is initially created. It checks whether the pool has already had its liquidity initialized and if it hasn't, it updates the liquidity. This function gets called from the Core.sol CreatePool function.

4. Function swap:

This function is used to swap a kind of token with the other. The function first checks whether the type of token the user is trying to swap is valid. Then it checks whether the output amount after fees is greater than zero.

Then if all conditions are met, it updates the state of the Smart Contract by updating the liquidity values. Finally it performs the token transfers.

5. addLiquidity:

This function is used to add liquidity in terms of both tokens to the pool. The function first checks whether the amount to be added to the pool is valid.

Then the function updates the state of the Smart Contract by updating the liquidity values and the liquidity shares of the user that is adding liquidity.

Finally the function performs the necessary token transfers from the liquidity provider to the contract.

6. removeLiquidity:

This function is used by liquidity providers that want to withdraw from the pool. The function first checks whether the user that called it is a valid liquidity provider. (i.e. they have enough shares)

Then the function updates the state of the Smart Contract by updating the liquidity values and the liquidity shares of the user that is withdrawing liquidity. The function also computes the reward that the liquidity provider will receive from their stakes.

Finally the function performs the necessary token transfers from the contract to the liquidity provider.

7. addLiquidityWithOneToken:

This function is used to add liquidity with just one token to the pool. The function first checks whether the token being added to the pool and the amount to be added to the pool is valid.

Then the function updates the state of the Smart Contract by updating the liquidity values and the liquidity shares of the user that is adding liquidity. The function transforms the token to be added into both tokens while keeping the total value of the tokens constant.

Finally the function performs the necessary token transfers from the liquidity provider to the contract.

DeFi.sol File:

1. Class Member Variables:

The class declares two structs to represent loans and shares respectively. It also keeps maps that map user addresses to their loans and shares. The class also keeps maps that map users to how much stakes and interest they have in both tokens.

The class also contains constant numeric values that get used during computations.

2. Function borrowTokenA:

This function takes as parameter how many tokens the user wants to borrow and the collateral they present. It then checks whether the user already

has a loan and whether the amount of collateral presented is sufficient to cover the loan.

If all conditions are met, the function then updates the state of the contract by creating a loan struct and adding it to the list of active loans. Then it updates the liquidity of both tokens in the pool using the loan and collateral amounts.

Finally the function transfers the borrowed tokens to the user and the collateral to the Smart Contract.

3. Function borrowTokenB:

This function takes as parameter how many tokens the user wants to borrow and the collateral they present. It then checks whether the user already has a loan and whether the amount of collateral presented is sufficient to cover the loan.

If all conditions are met, the function then updates the state of the contract by creating a loan struct and adding it to the list of active loans. Then it updates the liquidity of both tokens in the pool using the loan and collateral amounts.

Finally the function transfers the borrowed tokens to the user and the collateral to the Smart Contract.

4. Function repayLoan:

This function is used by a user to repay their loans either partially or fully. The function first checks if the user has any active loans and whether the amount the user wishes to repay is a valid amount.

If all requirements are met the function then computes how much collateral should be returned to the user. The function then updates the state of the Smart Contract by updating the status of the user's loan and the liquidity values of both tokens using the amount of borrowed tokens being repaid and the amount of collateral being returned to the user. The function then checks whether the loan has been fully repaid and removes it from the list of active loans if it has.

The function finally transfers the repaid tokens from the user to the contract and the returned collateral from the contract to the user.

5. Function getHealthFactor:

This function is used to compute the health factor of a given loan. This function is called from the liquidate function to determine whether a user should be allowed to liquidate a given loan.

6. Function liquidate:

This function is used by liquidators who wish to liquidate a given loan either completely or partially. The function first checks if a given loan exists and whether it is healthy.

In the case that the loan is unhealthy then the function proceeds with the liquidation. The function then computes how much collateral should be returned to the liquidator from how much of the loan is being covered.

The function then updates the state of the smart contract by updating the status of the loan being liquidated and removes it from the list of active loans if it has been completely covered. It also updates the liquidity of both tokens in the pool using how many tokens are being repaid and how much collateral is being given to the liquidator.

The function finally transfers the tokens being repaid from the liquidator to the Contract and the collateral being returned from the contract to the liquidator.

7. Function lendTokenA:

This function allows a user to lend an amount of tokens into the pool. The function first checks if the amount of tokens being lent is valid.

The function then updates the state of the smart contract by updating the liquidity of the token being lent in the pool and also the amount of stake the user has in the pool.

The function finally transfers the lent tokens from the user to the Contract.

8. Function lendTokenB:

This function allows a user to lend an amount of tokens into the pool. The function first checks if the amount of tokens being lent is valid.

The function then updates the state of the smart contract by updating the liquidity of the token being lent in the pool and also the amount of stake the user has in the pool.

The function finally transfers the lent tokens from the user to the Contract.

9. Function withdrawTokenA:

This function allows users to withdraw the tokens that they had lent to the pool. The function first checks if the user has a valid amount of stakes.

If the check is successful then the function computes the reward that the user should receive along with their withdrawn tokens using their stake. Then the function updates the state of the smart contract by decreasing the liquidity of the token by the amount being withdrawn and also decreasing the stake of the user in the pool.

The pool finally transfers the tokens being withdrawn from the Contract to the user.

10. Function withdrawTokenB:

This function allows users to withdraw the tokens that they had lent to the pool. The function first checks if the user has a valid amount of stakes.

If the check is successful then the function computes the reward that the user should receive along with their withdrawn tokens using their stake. Then the function updates the state of the smart contract by decreasing the liquidity of the token by the amount being withdrawn and also decreasing the stake of the user in the pool.

The pool finally transfers the tokens being withdrawn from the Contract to the user.

Core.sol File:

1. Class Member Variables:

The class keeps the addresses of all of the pools created through it in an array as a class member variable.

2. Function createPool:

This function takes the parameters necessary to call the constructor of the AMM class. It then deploys a pool using those parameters and initializes its liquidity. It then transfers the tokens from the user that deployed the pool to the pool contract that was created. Finally it adds the address of the pool that was just created to the array of pools.

Security:

Since our application is meant for Decentralized Finance user security is of utmost importance to our project.

While developing our project, we applied certain security principles, modules and tools to make sure user assets are sufficiently protected from security breaches.

Security Modules:

We use the SafeERC20 and ReentrancyGuard modules from OpenZeppelin to ensure a more secure implementation.

SafeERC20 is a ERC20 compliant token implementation that implements the safeTransfer() and safeTransferFrom() functions. These functions throw errors on failure and make sure that functions revert if the transfer fails.

Since we make our token transfers at the end of our function calls to comply with the Checks-Effects-Interactions principle, the use of the SafeTransfer functions make sure that any changes made to the contracts state will be reverted if the transfer fails.

Reentrancy Attack Prevention:

A Reentrancy Attack is a Smart Contract hack where a function of a Smart Contract is called multiple times in a single transaction to manipulate the Smart Contract more times than a user should be able to.

Such attacks are commonly used by attackers to steal tokens and other digital currencies by using withdraw or transfer functions of Smart Contracts multiple times.

Reentrancy attacks are performed by exploiting security vulnerabilities in Smart Contracts that allow a function to be called before a previous call to it has terminated.

If the function had not updated the state of the Smart Contract before it was called again, this can allow attackers to manipulate the behavior of the Smart Contract to perform malicious actions such as withdrawing or transferring tokens multiple times.

Reentrancy attacks are performed using attacker contracts that are deployed by malicious users.

The malicious user forces the victim contract to make an external call to their attacker contract from which they will call the victim contract again before the first call to it has completed.

The Checks-Effects-Interactions Pattern:

The Checks-Effects-Interactions pattern is a principle in Smart Contract programming where a function of a smart contract first performs necessary checks for the function to proceed, then updates the state of the smart contract and finally performs interactions with other smart contracts.

This pattern is used to protect the smart contract from security vulnerabilities and potential security breaches such as reentrancy attacks.

Since this pattern ensures that the state of the Smart Contract will be properly updated before any external calls are made, attackers will not be able to call the contract again before it has updated its state to manipulate its behavior.

The Checks-Effects-Interactions pattern was used as much as possible while implementing the smart contracts.

Use of ReentrancyGuard and Non Reentrant Functions

The use of the ReentrancyGuard module allows us to declare functions as non reentrant. A function that is declared as non reentrant cannot be called again while a previous call to it has not completed.

This ensures that malicious users cannot call a function multiple times before the state of the Smart Contract has been updated.

In the implementation of the De-Fi Application, another way in which potential reentrancy attacks are avoided is by avoiding external calls to other smart contracts. The only external function calls from the Smart Contracts are to the OpenZeppelin ERC20 transfer() and transferFrom() functions.

Since these functions themselves do not make any external function calls, malicious users cannot call a function multiple times through an attacker contract in a single transaction since they cannot force the Smart Contracts to make a call to their attacker contract.

Static Code Analysis

Static Code Analyzer plug-in of Remix was used to analyze the Smart Contracts for potential security issues.

When checked with the analyzer, the final implementation still returns warnings however these warnings do not actually indicate security flaws.

Static Code Analysis Warnings:

The following are explanations of the analyzer warnings that the project code has received and how they do not indicate security flaws.

1. Gas Cost Warning

This warning is raised when Remix is not able to compute a maximum gas cost for a function.

Since the contracts can be deployed and all the functions can be called on the Sepolia Testnet we can conclude that this warning is inaccurate.

2. Similar Variable Names Warning

This warning is raised when Remix considers variable names to be too similar to each other.

This warning is meant to encourage code readability and has no significance in regards to security.

3. Guard Conditions Warning

This warning is related to which guard condition should be used when checking for a requirement.

In our case we use `require(x)` and we check for user inputted values which can be false, hence the use of the guard conditions is correct.

4. Data Truncated Warning

This warning is to do with how results are computed during integer division and is not related to contract security.

Since Solidity does not support floating point numbers, we have to perform all our computations using integers.

5. Checks-Effects-Interactions Warning

This warning is an actual security concern because it detects a potential violation of Checks-Effects-Interactions which can lead to a reentrancy attack.

However after checking the flagged functions we see that the functions do not violate Checks-Effects-Interactions and that the warning is a false flag.

Checks-Effects-Interactions Warning Explanations:

The Checks-Effects-Interactions violation warning is raised for 6 functions all of which are in `DeFi.sol`. These functions are:

- DeFi.borrowTokenA()
- DeFi.borrowTokenB()
- DeFi.repayLoan()
- DeFi.liquidate()
- DeFi.lendTokenA()
- DeFi.lendTokenB()
- DeFi.withdrawTokenA()
- DeFi.withdrawTokenB()

The following code screenshots illustrate how these warnings are false flags.

Function borrowTokenA():

```
function borrowTokenA(uint256 collateralAmount, uint256 borrowAmount, address poolAddress) external nonReentrant {
    require(loans[msg.sender].borrowedAmount == 0, "Loan already active");

    AMM pool = AMM(poolAddress);
    address tokenA = pool.token0();
    address tokenB = pool.token1();
    uint256 liquidityA = pool.liquidity0();
    uint256 liquidityB = pool.liquidity1();

    uint256 collateralValueInA = (liquidityA * collateralAmount) / liquidityB;

    require(collateralValueInA * 100 >= borrowAmount * 150, "Insufficient collateral: min 150% required");
    require(totalTokenAStaked[poolAddress] >= borrowAmount, "Insufficient TokenA liquidity");

    loans[msg.sender] = Loan({
        collateralAmount: collateralAmount,
        borrowedAmount: borrowAmount,
        loanType: LoanType.TokenA
    });

    totalTokenAStaked[poolAddress] -= borrowAmount;
    totalTokenBStaked[poolAddress] += collateralAmount;

    IERC20(tokenB).safeTransferFrom(msg.sender, address(this), collateralAmount);
    IERC20(tokenA).safeTransfer(msg.sender, borrowAmount);

    emit LoanCreated(msg.sender, poolAddress, collateralAmount, borrowAmount, LoanType.TokenA);
    emit CollateralValueCalculated(msg.sender, poolAddress, collateralValueInA, liquidityA * 1e18 / liquidityB);
}
```

Checks

Effects

Interactions

Function borrowTokenB():

```
function borrowTokenB(uint256 collateralAmount, uint256 borrowAmount, address poolAddress) external nonReentrant {  
    require(loans[msg.sender].borrowedAmount == 0, "Loan already active");  
  
    AMM pool = AMM(poolAddress);  
    address tokenA = pool.token0();  
    address tokenB = pool.token1();  
    uint256 liquidityA = pool.liquidity0();  
    uint256 liquidityB = pool.liquidity1();  
  
    uint256 collateralValueInB = (liquidityB * collateralAmount) / liquidityA;  
  
    require(collateralValueInB * 100 >= borrowAmount * 150, "Insufficient collateral: min 150% required");  
    require(totalTokenBStaked[poolAddress] >= borrowAmount, "Insufficient TokenB liquidity");  
  
    loans[msg.sender] = Loan({  
        collateralAmount: collateralAmount,  
        borrowedAmount: borrowAmount,  
        loanType: LoanType.TokenB  
    });  
  
    totalTokenBStaked[poolAddress] -= borrowAmount;  
    totalTokenAStaked[poolAddress] += collateralAmount;  
  
    IERC20(tokenA).safeTransferFrom(msg.sender, address(this), collateralAmount);  
    IERC20(tokenB).safeTransfer(msg.sender, borrowAmount);  
  
    emit LoanCreated(msg.sender, poolAddress, collateralAmount, borrowAmount, LoanType.TokenB);  
    emit CollateralValueCalculated(msg.sender, poolAddress, collateralValueInB, liquidityB * 1e18 / liquidityA);  
}
```

Checks

Effects

Interactions

Function repayLoan():

```
function liquidate(address user, address poolAddress) external nonReentrant {  
    Loan storage loan = loans[user];  
    require(loan.borrowedAmount > 0, "No active loan");  
  
    uint256 healthFactor = getHealthFactor(user, poolAddress);  
    require(healthFactor < 1e18, "Health factor is healthy");  
  
    // Emit liquidation check event  
    emit LiquidationCheck(user, poolAddress, healthFactor, true);  
  
    uint256 repayAmount = (loan.borrowedAmount * LIQUIDATION_FACTOR_NUMERATOR) / LIQUIDATION_FACTOR_DENOMINATOR;  
    uint256 collateralToSeize = (loan.collateralAmount * LIQUIDATION_FACTOR_NUMERATOR) / LIQUIDATION_FACTOR_DENOMINATOR;  
  
    AMM pool = AMM(poolAddress);  
    address tokenA = pool.token0();  
    address tokenB = pool.token1();  
  
    if (loan.loanType == LoanType.TokenA) {  
        totalTokenAStaked[poolAddress] += repayAmount;  
        totalInterestTokenA[poolAddress] += (repayAmount * INTEREST_RATE_NUMERATOR) / INTEREST_RATE_DENOMINATOR;  
        emit InterestAccrued(poolAddress, tokenA, (repayAmount * INTEREST_RATE_NUMERATOR) / INTEREST_RATE_DENOMINATOR);  
    } else {  
        totalTokenBStaked[poolAddress] += repayAmount;  
        totalInterestTokenB[poolAddress] += (repayAmount * INTEREST_RATE_NUMERATOR) / INTEREST_RATE_DENOMINATOR;  
        emit InterestAccrued(poolAddress, tokenB, (repayAmount * INTEREST_RATE_NUMERATOR) / INTEREST_RATE_DENOMINATOR);  
    }  
  
    loan.borrowedAmount -= repayAmount;  
    loan.collateralAmount -= collateralToSeize;  
  
    if (loan.borrowedAmount == 0) {  
        delete loans[user];  
    }  
  
    if (loan.loanType == LoanType.TokenA) {  
        IERC20(tokenA).safeTransferFrom(msg.sender, address(this), repayAmount);  
        IERC20(tokenB).safeTransfer(msg.sender, collateralToSeize);  
    } else {  
        IERC20(tokenB).safeTransferFrom(msg.sender, address(this), repayAmount);  
        IERC20(tokenA).safeTransfer(msg.sender, collateralToSeize);  
    }  
  
    emit LoanLiquidated(user, poolAddress, msg.sender, repayAmount, collateralToSeize);  
  
    // Emit updated health factor after liquidation if loan still exists  
    if (loan.borrowedAmount > 0) {  
        uint256 newHealthFactor = getHealthFactor(user, poolAddress);  
        emit HealthFactorCalculated(user, poolAddress, newHealthFactor);  
    }  
}
```

Checks

Effects

Interactions

Function liquidate():

```
function repayLoan(address poolAddress, uint256 amount) external nonReentrant {
    Loan storage loan = loans[msg.sender];
    require(loan.borrowedAmount > 0, "No active loan");
    require(amount > 0 && amount <= loan.borrowedAmount, "Invalid repayment amount");

    AMM pool = AMM(poolAddress);
    address tokenA = pool.token0();
    address tokenB = pool.token1();

    uint256 previousBorrowedAmount = loan.borrowedAmount;
    uint256 amountInterest = amount * INTEREST_RATE_NUMERATOR / INTEREST_RATE_DENOMINATOR;

    if (loan.loanType == LoanType.TokenA) {
        totalTokenASTaked[poolAddress] += amount;
        totalInterestTokenA[poolAddress] += amountInterest;
        emit InterestAccrued(poolAddress, tokenA, amountInterest);
    } else {
        totalTokenBStaked[poolAddress] += amount;
        totalInterestTokenB[poolAddress] += amountInterest;
        emit InterestAccrued(poolAddress, tokenB, amountInterest);
    }

    loan.borrowedAmount -= amount;

    uint256 collateralReturn = (loan.collateralAmount * amount) / previousBorrowedAmount;
    loan.collateralAmount -= collateralReturn;

    if (loan.borrowedAmount == 0) {
        delete loans[msg.sender];
    }

    if (loan.loanType == LoanType.TokenA) {
        IERC20(tokenA).safeTransferFrom(msg.sender, address(this), amount + amountInterest);
        IERC20(tokenB).safeTransfer(msg.sender, collateralReturn);
    } else {
        IERC20(tokenB).safeTransferFrom(msg.sender, address(this), amount + amountInterest);
        IERC20(tokenA).safeTransfer(msg.sender, collateralReturn);
    }

    emit LoanRepaid(msg.sender, poolAddress, amount, amountInterest, collateralReturn);

    // Emit health factor after repayment if loan still exists
    if (loan.borrowedAmount > 0) {
        uint256 healthFactor = getHealthFactor(msg.sender, poolAddress);
        emit HealthFactorCalculated(msg.sender, poolAddress, healthFactor);
    }
}
```

Checks

Effects

Interactions

Function lendTokenA():

```
function lendTokenA(uint256 amount, address poolAddress) external nonReentrant {
    require(amount > 0, "Cannot lend 0");

    address tokenA = AMM(poolAddress).token0();
    totalTokenASTaked[poolAddress] += amount;

    shares[msg.sender][poolAddress].tokenA_share += amount;
    IERC20(tokenA).safeTransferFrom(msg.sender, address(this), amount);

    emit TokenLent(msg.sender, poolAddress, tokenA, amount);
}
```

Checks


Effects

Interactions


Function lendTokenB():

```
function lendTokenB(uint256 amount, address poolAddress) external nonReentrant {  
    require(amount > 0, "Cannot lend 0"); Checks  
  
    address tokenB = AMM(poolAddress).token1();  
    totalTokenBStaked[poolAddress] += amount; Effects  
  
    shares[msg.sender][poolAddress].tokenB_share += amount;  
    IERC20(tokenB).safeTransferFrom(msg.sender, address(this), amount); Interactions  
  
    emit TokenLent(msg.sender, poolAddress, tokenB, amount);  
}
```

Function withdrawTokenA():

```
function withdrawTokenA(address poolAddress) external nonReentrant {  infinite gas  
    Share storage share = shares[msg.sender][poolAddress];  
    uint256 userShare = share.tokenA_share;  
    require(userShare > 0, "Nothing to withdraw"); Checks  
  
    uint256 totalStaked = totalTokenAStaked[poolAddress];  
    uint256 interest = totalInterestTokenA[poolAddress];  
    uint256 userInterest = (userShare * interest) / totalStaked;  
  
    uint256 totalAmount = userShare + userInterest;  
  
    totalTokenAStaked[poolAddress] -= userShare;  
    totalInterestTokenA[poolAddress] -= userInterest;  
    share.tokenA_share = 0; Effects  
  
    address tokenA = AMM(poolAddress).token0();  
    IERC20(tokenA).safeTransfer(msg.sender, totalAmount); Interactions  
  
    emit TokenWithdrawn(msg.sender, poolAddress, tokenA, totalAmount, userShare, userInterest);  
}
```

Function withdrawTokenB():

```
function withdrawTokenB(address poolAddress) external nonReentrant {  infinite gas  
    Share storage share = shares[msg.sender][poolAddress];  
    uint256 userShare = share.tokenB_share;  
    require(userShare > 0, "Nothing to withdraw"); Checks  
  
    uint256 totalStaked = totalTokenBStaked[poolAddress];  
    uint256 interest = totalInterestTokenB[poolAddress];  
    uint256 userInterest = (userShare * interest) / totalStaked;  
  
    uint256 totalAmount = userShare + userInterest;  
  
    totalTokenBStaked[poolAddress] -= userShare;  
    totalInterestTokenB[poolAddress] -= userInterest;  
    share.tokenB_share = 0; Effects  
  
    address tokenB = AMM(poolAddress).token1();  
    IERC20(tokenB).safeTransfer(msg.sender, totalAmount); Interactions  
  
    emit TokenWithdrawn(msg.sender, poolAddress, tokenB, totalAmount, userShare, userInterest);  
}
```

As is visible from the screenshots above, the checks, effects and interactions components of the functions are all in the correct order. This means that the checks-effects-interactions pattern is not actually violated and that the analyzer warnings are false flags.

Project Github Repository Link:

<https://github.com/Borhanxj/defi-blockchain-app/tree/main>