KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: **KnownSec blockchain security research team**

Release notes

Revised content	Time	Revised by	Version number
Written document	20210121	KnownSec blockchain security research team	V1.0

Document information

Document name	Document version number	Document number	Confidentiality level
Box Smart Contract Audit Report	V1.0	4a332ae36a764183acd8d082250f5103	**Open project Team**

The statement

KnownSec only issues this report on the facts that have occurred or exist before the issuance of this report, and shall assume the corresponding responsibility therefor.KnownSec is not in a position to judge the security status of its smart contract and does not assume responsibility for the facts that occur or exist after issuance.The security audit analysis and other contents of this report are based solely on the documents and information provided by the information provider to KnownSec as of the issuance of this report.KnownSec assumes that the information provided was not missing, altered, truncated or suppressed.If the information provided is missing, altered, deleted, concealed or reflected in a way inconsistent with the actual situation, KnownSec shall not be liable for any loss or adverse effect caused thereby.

Directory

1. Review

The effective test time of this report is from January 14, 2021 to January 21, 2021. During this period, the security and standardization of the Box smart contract code will be audited and used as the statistical basis for the report.

In this test, KnownSec engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (see Chapter 3), and the comprehensive evaluation was passed.

The results of this smart contract security audit:Pass.

Since this testing process is carried out in a non-production environment, all codes are up-to-date backups, and the testing process is communicated with the relevant interface person, and relevant testing operations are carried out under the controllable operational risk to avoid production in the testing process Operational risk, code security risk.

Report information of this audit:

Report No:4a332ae36a764183acd8d082250f5103

Report query address link:

https://attest.im/attestation/searchresult?qurey=4a332ae36a764183acd8d082250f5103

Target information of this audit:

entry	description	
Token name	Box	
Code type	TRON Smart Contract	
Code language	Solidity	
Contract	Comptroller	TBfThHGPu78EapCnpLetyKBPZE2m48t2er

address	AssertPriceOracle	TQ5wiYRF68N9AwgK13tnM1uES2ydkAinqs
	JumpRateModel	THsxeUhV6qAZCPX4GejMW4KYNZbMxPRtn8
	CNative	TKyotncVJxFxztu7DTv3zdmqdTUsYsFtFR
	CErc20Delegator	TJJpqmQVozm6KbxsshjrMJkHahZ3AHfCuB
	CErc20Delegator	TG6z24Wz7uy5Qi8Skq8p3hzAidGSqGQymd
	CErc20Delegator	TYfFQsuBYgbhG5WX5K73fbjcvFZM7f3Wsd
	CErc20Delegator	TMhtKRofPqV1UdtVnrexR5kUuoXAEzHwRX
	CErc20Delegator	TM8uyte1PR8knWsVtboFFmRcJZ8ActSgS4
	CErc20Delegator	TLaEhK7UjEEr4Pf7DmXLNCX4adSbhn1tkw

Contract document and hash:

Contract documents	MD5
FeeManager.sol	C5F6B2EFE2EC91101911B77EC6261D23
CarefulMath.sol	6E307FB2C177144BFABFD26C9DEF29E8
Context.sol	EE8F37572F82A54CE72912BAD96343A1
Controller.sol	F3847D2C3ED47FBB8EF47F7A5D0B4050
EnumerableSet.sol	728294CE4C12208919076D861B172F18
ERC20.sol	FF18BF86C5A674DF6C4173E8AAB7566C
ERC20Burnable.sol	C4F5820744C45490CFD12B3A625E667C
ERC20Detailed.sol	A1A39233394F3C0A00CE348C61433FAA
ERC20Mintable.sol	D0828425DBB042C158558788C4BBC338
Exponential.sol	95B0C75925B3B337517ECCA57DB03AE5
IERC20.sol	183FF1F5E5F04BCFA85CEE597BCB0236
Math.sol	32FA90BA71DE78F2ADBE8A803D2D90B3

MinterRole.sol	2D9EC4DE161520693C92BD68EBADF363
NativeAddressLib.sol	648202C2B321BB5A7B8DD1B160E047E7
Ownable.sol	036CB43163C154F43DB99590E741E801
Roles.sol	118D44AA29245B854A8785A01AF0C4E9
SafeMath.sol	59D6A4434F7EAF874949A63A34C16B66
AssertPriceOracle.sol	B0A22C349F87E062F611E4C86B939C13
IRewardPool.sol	4D853C13000099FFFEA53B555F7AC85E
LPTokenWrapper.sol	0CE4424EB69028ED393E8535151E510F
Reservoir.sol	A4D53F03C465C07C113390F2B108AB6B
RewardPool.sol	08EA94731298E0D95A20B7B95E5B876B
RewardToken.sol	2A63E39CA8D584412CDE42301BD3A12B
BaseJumpRateModelV2.sol	407955E406690C5430F328AE709ADB73
CErc20.sol	BEFEDDD76F6DF03DC0CB03C685BB2E64
CErc20Delegate.sol	BC0D3DF68E0518CD4321991010762BBB
CErc20Delegator.sol	9A4603DE27C01987179D2EBEAD6EAD3E
CErc20Immutable.sol	A0B86D92E99899F710CCC322B25A536D
CNative.sol	1A25DB4701976684F713C58D333EBE5E
Comptroller.sol	0522C49F8B9858EF03BD9E1FF8A1DE7E
CToken.sol	AD93767AF10DCF651D9C516BD479D1D4
ErrorReporter.sol	66D35BE7A0050D7BE978A1E044966028
JumpRateModel.sol	4AA25F4B57B6044E2F203E582E5B44E5

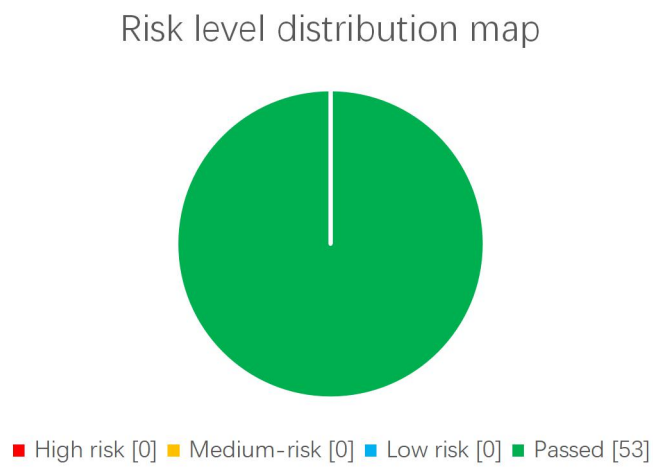
JumpRateModelV2.sol	B58D68DE521A209AC770027FF490A3C3
LegacyJumpRateModelV2.sol	430C924A87809CCB44927608E1857387
Maximillion.sol	037442A306408EF0BAC9B549E361C19A
Migrations.sol	AEB82FB35EFAC0EC173CD1376279CE67
RiskController.sol	8C34694F39172DD5CBBE7E056AA67F20
Unitroller.sol	9BC51EF9B26485B358E8330F194462D2
WhitePaperInterestRateModel.sol	10C844936BF1C848752D37FE13F36A40

2. Code vulnerability analysis

2.1 Vulnerability level distribution

This vulnerability risk is calculated by level:

Statistics on the number of security risk levels			
High Risk	Medium Risk	Low Risk	Pass
0	0	0	53



2.2 Summary of audit results

Audit results			
Audit item	Audit content	status	description
Business security testing	SwapToken logic design	pass	After testing, there are no safety issues.
	addTokenPoolLiquidity logic design	pass	After testing, there are no safety issues.
	SetBasePoolToken logic design	pass	After testing, there are no safety issues.
	SetUnderlyingOracle logic design	pass	After testing, there are no safety issues.
	GetUnderlyingPrice logic design	pass	After testing, there are no safety issues.
	Stake pledge logic design	pass	After testing, there are no safety issues.
	Withdraw logic design	pass	After testing, there are no safety issues.
	MintToken logic design	pass	After testing, there are no safety issues.
	Increase liquidity pool logic design	pass	After testing, there are no safety issues.
	DistributionToken logic design	pass	After testing, there are no safety issues.
	ClaimFund logic design	pass	After testing, there are no safety issues.
	Exit exit logic design	pass	After testing, there are no safety issues.
	NotifyRewardAmount logic design	pass	After testing, there are no safety issues.
	UpdateJumpRateModel logic design	pass	After testing, there are no safety issues.
	getBorrowRateInternal logic design	pass	After testing, there are no safety issues.
	Mint logic design	pass	After testing, there are no safety issues.
	Redeem logic design	pass	After testing, there are no safety issues.

	Borrow logic design	pass	After testing, there are no safety issues.
	repayBorrow logic design	pass	After testing, there are no safety issues.
	LiquidityBorrow logic design	pass	After testing, there are no safety issues.
	_addReserves logic design	pass	After testing, there are no safety issues.
	_reduceReserves logic design	pass	After testing, there are no safety issues.
	isConstract logic design	pass	After testing, there are no safety issues.
Basic code vulnerability detection	Reentry attack detection	pass	After testing, there are no safety issues.
	Replay attack detection	pass	After testing, there are no safety issues.
	Rearrangement attack detection	pass	After testing, there are no safety issues.
	Numerical overflow detection	pass	After testing, there are no safety issues.
	Arithmetic accuracy error	pass	After testing, there are no safety issues.
	Access control defect detection	pass	After testing, there are no safety issues.
	tx.origon authentication	pass	After testing, there are no safety issues.
	call injection attack	pass	After testing, there are no safety issues.
	Return value call verification	pass	After testing, there are no safety issues.
	Uninitialized storage pointer	pass	After testing, there are no safety issues.
	Wrong use of random number detection	pass	After testing, there are no safety issues.
	Transaction order dependency detection	pass	After testing, there are no safety issues.
	Denial of service attack detection	pass	After testing, there are no safety issues.
	Fake recharge vulnerability detection	pass	After testing, there are no safety issues.
	Additional token issuance	pass	After testing, there are no safety issues.

	vulnerability detection		
	Frozen account bypass detection	pass	After testing, there are no safety issues.
	Compiler version security	pass	After testing, there are no safety issues.
	Not recommended encoding	pass	After testing, there are no safety issues.
	Redundant code	pass	After testing, there are no safety issues.
	Use of safe arithmetic library	pass	After testing, there are no safety issues.
	Use of require/assert	pass	After testing, there are no safety issues.
	Energy consumption detection	pass	After testing, there are no safety issues.
	fallback function safety	pass	After testing, there are no safety issues.
	Owner permission control	pass	After testing, there are no safety issues.
	Low-level function safety	pass	After testing, there are no safety issues.
	Variable coverage	pass	After testing, there are no safety issues.
	Timestamp dependent attack	pass	After testing, there are no safety issues.
	Unsafe interface use	pass	After testing, there are no safety issues.

3. Business security testing

3.1. SwapToken logic design[Pass]

Audit the swapToken token swap logic design, check whether there is a check on the parameters, the rationality of the logic design, and the authority check of the function caller.

Audit result: The swapToken function checks the validity of the parameters, and restricts the function caller's authority through onlyController, and no security risks are found in the related logic design.

```
function swapToken(
        address poolAddress, address tokenIn, address tokenOut
) onlyController public returns (uint){
    require(factory != address(0), "factory=0");//knownsec// Address non-zero check
    require(IFactory(factory).isBPool(poolAddress), "!pool");//knwonsec// Pool address check
    uint _tokenInAmount = IERC20(tokenIn).balanceOf(address(this));//knownsec// Calculate the number of tokens to be exchanged
    IERC20(tokenIn).approve(poolAddress, _tokenInAmount);//knownsec// Authorized transfer operation
    (uint tokenAmountOut,)=IBool(poolAddress).swapExactAmountIn(tokenIn, _tokenInAmount, tokenOut, 0, MAX);//knownsec// Exchange operation
        return tokenAmountOut;//knownsec// Returns the number of expected tokens that can be exchanged
}
modifier onlyController() {
        require(isController(msg.sender), "!controller");//knownsec// Permission check
        _;
}
```

Safety advice: none.

3.2. addTokenPoolLiquidity logic design[Pass]

Audit the business logic of addTokenPoolLiquidity to check whether there is a check on the legality of the parameters and the rationality of the logic design, whether there is an overflow risk, etc.

Audit result: addTokenPoolLiquidity has checked the legality of the parameters and used SafeMath to perform numerical operations. There is no risk of overflow, and the relevant logic design is correct and no safety risk is found.

```
function addTokenPoolLiquidity() external {
        require(IERC20(boxToken).balanceOf(msg.sender) >= dexTokenAmount, "not enough token");//knwonsec// token quantity check
        uint _amount = IERC20(baseToken).balanceOf(address(this));
        uint256 _goverFund = _amount.mul(goverFundDivRate).div(1e18);//knownsec// Calculate the number of governorFund
        IERC20(baseToken).transfer(governaddr, _goverFund);//knownsec// Transfer _goverFund amount of tokens to governAddr

        uint _tokenInAmount = _amount.mul(burnRate).div(1e18);
        IERC20(baseToken).approve(basePool, _tokenInAmount);
        IBool(basePool).swapExactAmountIn(baseToken, _tokenInAmount, boxToken, 0, MAX);
        //knownsec// Update _amount
        _amount = _amount.sub(_goverFund);
        _amount = _amount.sub(_tokenInAmount);
        IERC20(baseToken).approve(basePool, _amount);
        IBool(basePool).joinswapExternAmountIn(baseToken, _amount, 0);
    }
```

Safety advice: none.

3.3. SetBasePoolToken logic design[Pass]

Audit the business logic of setBasePoolToken to check whether there is a check on the legality of the parameters, whether the related logic design is reasonable, and whether the permission design is reasonable, etc.

Audit result: The setBasePoolToken function checks the identity of the function caller through onlyController, and checks the validity of the parameters through require. The relevant logic design is reasonable and correct.

```
    function setBasePoolToken(address _pool, address _token, address _baseToken) onlyController public {
        //knownsec// Related parameter check
        require(factory != address(0), "factory is 0");
        require(IFactory(factory).isBPool(_pool), "!pool");
        require(IBool(_pool).isBound(_token), "not bound");
        require(IBool(_pool).isBound(_baseToken), "not bound");
        require(setBasePoolCount < 2, "limit set base pool");
        setBasePoolCount = setBasePoolCount.add(1);
        basePool = _pool;
        boxToken = _token;
        baseToken = _baseToken;
    }
```

Safety advice: none.

3.4. SetUnderlyingOracle logic design[Pass]

Audit the logic design of setUnderlyingOracle to check whether the permission

design is reasonable and whether there is a security risk in the logic design.

Audit result: setUnderlyingOracle checks the identity of the caller through the decorator onlyController, and the related logic design is correct.

```
function setUnderlyingOracle(address underlying, address oracle) onlyController external {
        ILinkOracle(oracle).latestAnswer();
        underlyingOracles[underlying] = oracle;
    }
    //knownsec// The controller is initialized in the constructor
    constructor(address _controller, address _wtrxAddress) Controller() public {
            setController(_controller);
            wtrxAddress = _wtrxAddress;
    }
    //knownsec// setContrller logic
    function setController(address controller) public onlyOwner {
            require(controller != address(0), "controller is empty");
            emit SetController(controller, _controller);
            _controller = controller;
     }
```

Safety advice: none.

3.5. GetUnderlyingPrice logic design[Pass]

Audit the logic design of getUnderlyingPrice to check whether the parameters are checked and whether there are errors in the logic design.

Audit results: No related security risks were found.

```
function getUnderlyingPrice(CToken cToken) external view returns (uint){
        address _underlying = address(CErc20(address(cToken)).underlying());
        if (justLps[_underlying] == true) {
```

```
                return calJustLpPrice(_underlying);//knownsec// Call calJustLpPrice
            }
            if (abeloLps[_underlying] == true) {
                return calAbeloLpPrice(_underlying);//knownsec// Call calAbeloLpPrice
            }
            address oracle = underlyingOracles[_underlying];
            require(oracle != address(0), "oracle is 0");//knownsec// Address non-zero
judgment
            return toUint(ILinkOracle(oracle).latestAnswer());
        }
        //knownsec//calJustLpPrice Logic design and implementation
    function calJustLpPrice(address _pairAddress) public view returns (uint){
        address _underlying = justLpPairs[_pairAddress];
        require(underlyingOracles[_underlying] != address(0), "oracle is 0");//knownsec//
Check if the underlying exists
        uint _totalSupply = IJustPair(_pairAddress).totalSupply().div(10 **
IJustPair(_pairAddress).decimals());//knownsec// Calculate totalsupply
          uint _totalUnderlying = ERC20Detailed(_underlying).balanceOf(_pairAddress);
          uint _totalTrx = _pairAddress.balance;
          uint _underlyingDecimal = ERC20Detailed(_underlying).decimals();
          uint _lpPerUnderlying = _totalUnderlying.div(_totalSupply).div(10 **
_underlyingDecimal);
          uint _lpPerTrx = _totalTrx.div(_totalSupply).div(10 ** wtrxDecimals);
          address oracle = underlyingOracles[_underlying];
          uint _underLyingPrice = uint(ILinkOracle(oracle).latestAnswer());
          address wtrxOracle = underlyingOracles[wtrxAddress];
          uint _trxPrice = uint(ILinkOracle(wtrxOracle).latestAnswer());
          return _lpPerUnderlying.mul(_underLyingPrice).add(_lpPerTrx.mul(_trxPrice));
        }
        //knownsec//calAbeloLpPrice Logic design and implementation
      function calAbeloLpPrice(address _lpAddress) public view returns (uint){
          address _poolView = abeloLpViews[_lpAddress];
          uint total;
```

```
        address[] memory tokens = IPoolView(_poolView).getCurrentTokens();
        for (uint i = 0; i < 2; i++) {
            address token = tokens[i];
            uint _lpPerToken = IPoolView(_poolView).getTokenAmountPerLp(token);
            address oracle = underlyingOracles[token];
            uint _tokenPrice = uint(ILinkOracle(oracle).latestAnswer());
            total = total.add(_lpPerToken.mul(_tokenPrice));
        }
        return total;
    }
```

Safety advice: none.

3.6. Stake pledge logic design[Pass]

Audit the stake pledge logic to check the rationality of the logic design and whether there is an overflow risk.

Audit result: Stake adopts SafeMath to perform numerical operations, which is relatively safe, and the logical design is reasonable, and no related safety risks have been found.

```
function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        lpToken.transferFrom(msg.sender, address(this), amount);
}
```

Safety advice: none.

3.7. Withdraw logic design[Pass]

Audit the logic of withdraw withdrawal to check the rationality of the logic

design and whether there is an overflow risk.

Audit result: SafeMath is used for withdraw to perform numerical calculation operations, which is safer, the logic design is reasonable, and no related safety risks are found

```
    function withdraw(uint256 amount) public {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
         lpToken.transfer(msg.sender, amount);
     }
```

Safety advice: none.

3.8. MintToken logic design[Pass]

Audit the mintToken logic, check the rationality of the logic design, whether there is an overflow risk, whether the authority design is reasonable, etc.

Audit result: mintToken is modified by the modifier onlyOwner, and can only be called by the owner of the contract. MintToken can only be minted once and cannot be issued indefinitely. At the same time, SafeMath function is used for numerical operations during minting, and there is no risk of overflow.

```
    function mintToken() onlyOwner public {
            rewardToken.mint(address(this));//knownsec// Call mint function to realize minting
    }
    function mint(address _to) public onlyOwner {
        require(totalSupply() == 0, "mint once");//knownsec// Can only mint once
        _mint(_to, _totalSupply);
    }
```

```
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");
        _totalSupply = _totalSupply.add(amount);//knownsec// Increase total
        _balances[account] = _balances[account].add(amount);//knownsec// Increase
token
        emit Transfer(address(0), account, amount);
    }
```

Safety advice: none.

3.9. Increase liquidity pool logic design[Pass]

Audit the logic of increasing the liquidity pool, check whether the parameters are checked, whether the authority design is reasonable, whether the logic design is reasonable, etc.

Audit result: After testing, it was found that the authority was checked, and when the liquidity pool was added, it was judged whether the current pool already exists, and the related logic design was correct.

```
    function add(
        address _contractAddress,
        uint256 _allocPoint
    ) public onlyController adjustProduct {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo memory pool = poolInfo[pid];
            require(pool.contractAddress != _contractAddress, "contract is
exist");//knownsec// Determine whether the current liquidity pool already exists
        }
        uint256 lastRewardBlock = block.number > startBlock
        ? block.number
```

```
                : startBlock;
                totalAllocPoint = totalAllocPoint.add(_allocPoint);//knownsec//更新
                poolInfo.push(//knownsec// Add new LP logic
                    PoolInfo(
                    {
                    allocPoint : _allocPoint,
                    contractAddress : _contractAddress
                    }
                    )
                );
    }
    modifier onlyController() {
        require(isController(msg.sender), "!controller");//knownsec// Permission check
            _;
    }
    function isController(address account) public view returns (bool) {
            return account == _controller;
    }
    modifier adjustProduct()    {//knownsec// Adjustment processing
            _;
            if (block.number >= periodEndBlock) {
                if (tokenPerBlock > MIN_TOKEN_REWARD) {
                    tokenPerBlock = tokenPerBlock.mul(90).div(100);
                }
                if (tokenPerBlock < MIN_TOKEN_REWARD) {
                    tokenPerBlock = MIN_TOKEN_REWARD;
                }
                periodEndBlock = block.number.add(duration);// End of update cycle
block
            }}
```

Safety advice: none.

3.10. DistributionToken logic design[Pass]

Audit the distributionToken logic to check whether the parameters are verified and whether the logic design is reasonable.

Audit results: no relevant security risks were found.

```
function distributionToken() adjustProduct public {
        if (block.number < startBlock) {//knownsec// Determine whether it is currently in a cycle
            return;
        }
        if (poolLastRewardBlock >= periodEndBlock) {//knownsec// Judge whether it is over
            return;
        }
        uint multiplier = 1;
        if (periodEndBlock > block.number) {
            multiplier = periodEndBlock - block.number;//knownsec// Calculate how many blocks remain at the end of a period
        }
        uint _reward = multiplier.mul(tokenPerBlock);
      //knownsec// Proportionately
        uint goverFund = calRate(_reward, goverFundDivRate);
        uint insuranceFund = calRate(_reward, insuranceFundDivRate);
        uint teamFund = calRate(_reward, teamFundDivRate);
        uint liquidityFund = calRate(_reward, liquidityRate);

        uint distriReward = _reward.sub(goverFund.add(insuranceFund).add(teamFund).add(liquidityFund));

        safeTokenTransfer(address(comptroller), distriReward);
        Comptroller(comptroller)._setCompRate(calRate(tokenPerBlock,
```

```
comptrollerRate));
                //knownsec// The pool is divided into
                uint length = poolInfo.length;
                for (uint pid = 0; pid < length; ++pid) {
                    PoolInfo storage pool = poolInfo[pid];
                    uint256 poolReward = liquidityFund.mul(pool.allocPoint).div(totalAllocPoint);
                    safeTokenTransfer(pool.contractAddress, poolReward);
                    IRewardPool(pool.contractAddress).notifyRewardAmount(poolReward);
                }
                poolLastRewardBlock = periodEndBlock;
            }
            // Safe pickle transfer function, just in case if rounding error causes pool to not have enough dex.
            //knownsec// Transfer logic
            function safeTokenTransfer(address _to, uint256 _amount) internal {
                uint256 pickleBal = rewardToken.balanceOf(address(this));
                if (_amount > pickleBal) {
                    rewardToken.transfer(_to, pickleBal);
                } else {
                    rewardToken.transfer(_to, _amount);
                }
            }
```

Safety advice: none.

3.11. ClaimFund logic design[Pass]

Audit the claimFund logic to check whether the parameters are verified and whether the logic design is reasonable.

Audit result: claimFund has checked the legality of the parameters, and the related logic design is correct.

```
        // Distribute tokens according to teh speed of the block
        function claimFund() onlyController external {
            if (block.number <= fundLastRewardBlock) {//knownsec// Check if the last
block is reached
                return;
            }
            uint256 multiplier = block.number - fundLastRewardBlock;
            uint256 boxReward = multiplier.mul(tokenPerBlock);//knownsec// Calculate
reward
            uint goverFund = calRate(boxReward, goverFundDivRate);
            uint insuranceFund = calRate(boxReward, insuranceFundDivRate);
            uint teamFund = calRate(boxReward, teamFundDivRate);
            rewardToken.transfer(governaddr, goverFund);
            rewardToken.transfer(insuranceaddr, insuranceFund);
            uint256 perDevFund = teamFund.div(teamAddrs.length());//knownsec// team
is divided into
            for (uint256 i = 0; i < teamAddrs.length() - 1; i++) {
                rewardToken.transfer(teamAddrs.get(i), perDevFund);
            }
            uint256 remainFund = teamFund.sub(perDevFund.mul(teamAddrs.length() -
1));
            rewardToken.transfer(teamAddrs.get(teamAddrs.length() - 1), remainFund);
            goverFund = 0;
            insuranceFund = 0;
            teamFund = 0;
            fundLastRewardBlock = block.number;
            distributionToken();
        }
```

Safety advice: none.

3.12. Exit exit logic design[Pass]

Audit the exit logic, check whether the logic design is reasonable, whether the parameters are checked, etc.

Audit results: no relevant security risks were found.

```
function exit() external {
        withdraw(balanceOf(msg.sender));//knownsec// Withdrawal logic
        getReward();//knownsec// Withdraw reward
}
 function withdraw(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");//knownsec// Asset quantity check
        uint exitAmount = amount.mul(exitFee).div(1e18);//knownsec// exitfee calculation
        uint _amount = amount.sub(exitAmount);//knownsec// Withdrawable assets
        super.withdraw(_amount);//knownsec// Withdraw assets
        emit Withdrawn(msg.sender, amount);
}
  //knownsec// Calculate reward
  function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
         if (reward > 0) {//knownsec// Check whether reward is greater than 0
              rewards[msg.sender] = 0;
              // If it is a normal user and not smart contract,
              // then the requirement will pass
              // If it is a smart contract, then
              // make sure that it is not on our greyList.
              if (tx.origin == msg.sender) {//knownsec// Caller identity check
                   rewardToken.transfer(msg.sender, reward);//knownsec// extract
                   emit RewardPaid(msg.sender, reward);
              } else {
                   emit RewardDenied(msg.sender, reward);
```

```
                }
            }
    }
    //knownsec// Income calculation
     function earned(address account) public view returns (uint256) {
            return
            balanceOf(account)
            .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
            .div(1e18)
            .add(rewards[account]);
      }
```

Safety advice: none.

3.13. NotifyRewardAmount logic design[Pass]

Audit the notifyRewardAmount logic to check whether the legality of the parameters is checked, whether the logic design is reasonable, and the permission design is reasonable.

Audit result: notifyRewardAmount checks the identity of the caller and also checks the parameters. The function logic design is correct.

```
    function notifyRewardAmount(uint256 reward)
        external
        updateReward(address(0))
        {
            require(msg.sender == controller() || msg.sender == reservoirAddress, "only controller and reservoir");//knownsec// Permission check

            require(reward < uint(- 1) / 1e18, "the notified reward cannot invoke multiplication overflow");//knownsec// Overflow check
```

```
            if (block.number >= periodFinish) {
                rewardRate = reward.div(duration);
            } else {
                uint256 remaining = periodFinish.sub(block.number);
                uint256 leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(duration);
            }
            lastUpdateTime = block.number;//knownsec// Update lastUPdateTime
            periodFinish = block.number.add(duration);//knownsec// Update Cycle
            emit RewardAdded(reward);
        }
```

Safety advice: none.

3.14. UpdateJumpRateModel logic design[Pass]

Audit the parameter function of updateJumpRateModel to update the interest rate model to check whether there is a permission check on the identity of the caller and whether there is a security risk in the logic design.

Audit result: updateJumpRateModel checks the identity of the caller.

```
    function updateJumpRateModel(
            uint baseRatePerYear,
            uint multiplierPerYear,
            uint jumpMultiplierPerYear,
            uint kink_
        ) external {
            require(msg.sender == owner, "only the owner may call this function.");//knownsec// Permission check on the identity of the caller
            updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
```

```
        }
    //knownsec// Update interest rate model parameters
    function updateJumpRateModelInternal(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_
    ) internal {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = (multiplierPerYear.mul(1e18)).div(blocksPerYear.mul(kink_));
        jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
        kink = kink_;
        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
    }
```

Safety advice: none.

3.15. getBorrowRateInternal logic design[Pass]

Audit the borrowing rate logic of getBorrowRateInternal to check whether the parameters are verified, etc.

Audit results: no relevant security risks were found.

```
    function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
        uint util = utilizationRate(cash, borrows, reserves);
        if (util <= kink) {
            return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
        } else {
            uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
```

```
            uint excessUtil = util.sub(kink);
            return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
        }
    }
    //knownsec// Calculate market interest rates
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint)
{
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }
        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }
```

Safety advice: none.

3.16. getSupplyRate logic design[Pass]

Audit the logic of getSupplyRate to check for overflow risks and logic design errors.

Audit result: getSupplyRate uses SafeMath to perform numerical operations, there is no risk of overflow, and the related logic design is correct.

```
function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRateInternal(cash, borrows, reserves);
```

```
            uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
            return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
}
    //knownsec// Borrowing rate
    function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
            uint util = utilizationRate(cash, borrows, reserves);

            if (util <= kink) {
                return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            } else {
                uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
                uint excessUtil = util.sub(kink);
                return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
            }
    }
```

Safety advice: none.

3.17. mint logic design[Pass]

Audit the mint logic design, check whether the parameters are verified, whether the function caller permissions are checked, whether there are errors in the logic design, etc.

Audit result: The logic design of mint coin function is correct, and there is a clear authority check.

```
    function mint(uint mintAmount) external returns (uint) {
        (uint err,) = mintInternal(mintAmount);
        return err;
    }
```

```
    function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {//knownsec// Non-empty check
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');//knownsec// risk check
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec// Error type checking
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
        }
        // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need to
        return mintFresh(msg.sender, mintAmount);

    }
    //knownsec//mintFresh Logical design
    function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
        /* Fail if mint not allowed */
        uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);//knownsec// Permission check
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
```

```
        }

        MintLocalVars memory vars;

        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr)), 0);
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         *  We call `doTransferIn` for the minter and the mintAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  `doTransferIn` reverts if anything goes wrong, since we can't be sure if
         *  side-effects occurred. The function returns the amount actually transferred,
         *  in case of a fee. On success, the cToken holds an additional `actualMintAmount`
         *  of cash.
         */
        vars.actualMintAmount = doTransferIn(minter, mintAmount);

        /*
         * We get the current exchange rate and calculate the number of cTokens to be minted:
         *  mintTokens = actualMintAmount / exchangeRate
         */

        (vars.mathErr, vars.mintTokens) = divScalarByExpTruncate(
```

```
                vars.actualMintAmount,
                Exp({mantissa : vars.exchangeRateMantissa})
            );
            require(vars.mathErr == MathError.NO_ERROR,
"MINT_EXCHANGE_CALCULATION_FAILED");

            /*
             * We calculate the new total supply of cTokens and minter token balance,
checking for overflow:
             *    totalSupplyNew = totalSupply + mintTokens
             *    accountTokensNew = accountTokens[minter] + mintTokens
             */
            (vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens);
            require(vars.mathErr == MathError.NO_ERROR,
"MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");

            (vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter],
vars.mintTokens);
            require(vars.mathErr == MathError.NO_ERROR,
"MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");

            /* We write previously calculated values into storage */
            totalSupply = vars.totalSupplyNew;
            accountTokens[minter] = vars.accountTokensNew;

            /* We emit a Mint event, and a Transfer event */
            emit Mint(minter, vars.actualMintAmount, vars.mintTokens);
            emit Transfer(address(this), minter, vars.mintTokens);

            /* We call the defense hook */
            comptroller.mintVerify(address(this), minter, vars.actualMintAmount,
vars.mintTokens);
            return (uint(Error.NO_ERROR), vars.actualMintAmount);
```

```
}
```

Safety advice: none.

3.18. redeem logic design[Pass]

Audit the redeem logic design, check whether the logic design is reasonable, whether the parameters are checked, etc.

Audit results: no relevant security risks were found.

```
function redeem(uint redeemTokens) external returns (uint) {
        return redeemInternal(redeemTokens);
    }
    function redeemInternal(uint redeemTokens) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec// Error checking
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem failed
            return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
        }
        // redeemFresh emits redeem-specific logs on errors, so we don't need to
        return redeemFresh(msg.sender, redeemTokens, 0);
    }
    //knownsec// Apply accrued interest to the total amount of borrowings and reserves, which will calculate the accumulated interest from the last checkpoint block to the current block, and write the new checkpoint to the memory
    function accrueInterest() public returns (uint) {
        /* Remember the initial block number */
        uint currentBlockNumber = getBlockNumber();
        uint accrualBlockNumberPrior = accrualBlockNumber;
```

```
        /* Short-circuit accumulating 0 interest */
        if (accrualBlockNumberPrior == currentBlockNumber) {
            return uint(Error.NO_ERROR);
        }

        /* Read the previous values out of storage */
        uint cashPrior = getCashPrior();
        uint borrowsPrior = totalBorrows;
        uint reservesPrior = totalReserves;
        uint borrowIndexPrior = borrowIndex;

        /* Calculate the current borrow interest rate */
        uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior, reservesPrior);
        require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");

        /* Calculate the number of blocks elapsed since the last accrual */
        (MathError mathErr, uint blockDelta) = subUInt(currentBlockNumber, accrualBlockNumberPrior);
        require(mathErr == MathError.NO_ERROR, "could not calculate block delta");

        /*
         * Calculate the interest accumulated into borrows and reserves and the new index:
         *  simpleInterestFactor = borrowRate * blockDelta
         *  interestAccumulated = simpleInterestFactor * totalBorrows
         *  totalBorrowsNew = interestAccumulated + totalBorrows
         *  totalReservesNew = interestAccumulated * reserveFactor + totalReserves
         *  borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
         */

        Exp memory simpleInterestFactor;
        uint interestAccumulated;
```

```
        uint totalBorrowsNew;
        uint totalReservesNew;
        uint borrowIndexNew;

        (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa : borrowRateMantissa}), blockDelta);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
```

```
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(
                Exp({mantissa : reserveFactorMantissa}),
                interestAccumulated,
                reservesPrior
            );
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,

FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor,
borrowIndexPrior, borrowIndexPrior);
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,

FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
                    uint(mathErr)
                );
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)
```

```
        /* We write the previously calculated values into storage */
        accrualBlockNumber = currentBlockNumber;
        borrowIndex = borrowIndexNew;
        totalBorrows = totalBorrowsNew;
        totalReserves = totalReservesNew;

        /* We emit an AccrueInterest event */
        emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew);

        return uint(Error.NO_ERROR);
    }
  //knownsec//redeemFresh Logical design
    function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal returns (uint) {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");
        RedeemLocalVars memory vars;
        /* exchangeRate = invoke Exchange Rate Stored() */
        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr));
        }
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
```

```
            vars.redeemTokens = redeemTokensIn;

            (vars.mathErr, vars.redeemAmount) = mulScalarTruncate(
                Exp({mantissa : vars.exchangeRateMantissa}), redeemTokensIn
            );
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            (vars.mathErr, vars.redeemTokens) = divScalarByExpTruncate(
                redeemAmountIn,
                Exp({mantissa : vars.exchangeRateMantissa})
            );
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }
            vars.redeemAmount = redeemAmountIn;
        }
```

```
        /* Fail if redeem not allowed */
        uint allowed = comptroller.redeemAllowed(address(this), redeemer, vars.redeemTokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REDEEM_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDEEM_FRESHNESS_CHECK);
        }

        /*
         * We calculate the new total supply and redeemer balance, checking for underflow:
         *  totalSupplyNew = totalSupply - redeemTokens
         *  accountTokensNew = accountTokens[redeemer] - redeemTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = subUInt(totalSupply, vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

        (vars.mathErr, vars.accountTokensNew) = subUInt(accountTokens[redeemer], vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
```

```
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            /* Fail gracefully if protocol has insufficient cash */
            if (getCashPrior() < vars.redeemAmount) {
                return fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.REDEEM_TRANSFER_OUT_NOT_POSSIBLE);
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)

            /*
             * We invoke doTransferOut for the redeemer and the redeemAmount.
             *  Note: The cToken must handle variations between ERC-20 and ETH
underlying.
             *  On success, the cToken has redeemAmount less of cash.
             *  doTransferOut reverts if anything goes wrong, since we can't be sure if side
effects occurred.
             */
            uint _amount = vars.redeemAmount;
            uint _feeAmount = div_(mul_(_amount, redeemFee), expScale);
            totalReserves = add_(totalReserves, _feeAmount);
            _amount = sub_(_amount, _feeAmount);
            doTransferOut(redeemer, vars.redeemAmount);

            /* We write previously calculated values into storage */
```

```
            totalSupply = vars.totalSupplyNew;
            accountTokens[redeemer] = vars.accountTokensNew;

            /* We emit a Transfer event, and a Redeem event */
            emit Transfer(redeemer, address(this), vars.redeemTokens);
            emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

            /* We call the defense hook */
            comptroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.redeemTokens);

            return uint(Error.NO_ERROR);
        }
```

Safety advice: none.

3.19. borrow logic design[Pass]

Audit the logical design of borrow assets to check whether the parameters are legally verified and the logical design is reasonable.

Audit results: The borrow logic design is correct, and no related security risks are found.

```
    function borrow(uint borrowAmount) external returns (uint) {
        return borrowInternal(borrowAmount);
    }
    function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {//knownsec// Address check
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');//knownsec// Risk check
```

```
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            return fail(Error(error), FailureInfo.BORROW_ACCRUE_INTEREST_FAILED);
        }
        // borrowFresh emits borrow-specific logs on errors, so we don't need to
        return borrowFresh(msg.sender, borrowAmount);
    }
    //knownsec//borrowFresh Logical design
    function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }
```

```
        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         *    accountBorrowsNew = accountBorrows + borrowAmount
         *    totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows,
borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAIL
ED,
                uint(vars.mathErr)
            );
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
```

```
                    Error.MATH_ERROR,

FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)

            /*
             * We invoke doTransferOut for the borrower and the borrowAmount.
             *     Note: The cToken must handle variations between ERC-20 and ETH underlying.
             *    On success, the cToken borrowAmount less of cash.
             *    doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
             */
            doTransferOut(borrower, borrowAmount);

            /* We write the previously calculated values into storage */
            accountBorrows[borrower].principal = vars.accountBorrowsNew;
            accountBorrows[borrower].interestIndex = borrowIndex;
            totalBorrows = vars.totalBorrowsNew;

            /* We emit a Borrow event */
            emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

            /* We call the defense hook */
            comptroller.borrowVerify(address(this), borrower, borrowAmount);
```

```
        return uint(Error.NO_ERROR);
    }
```

Safety advice: none.

3.20. repayBorrow logic design[Pass]

Audit the logic design of repayBorrow repayment, check the rationality of the logic design and whether the parameters are checked.

Audit results: No related security risks were found.

```
    function repayBorrow(uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowInternal(repayAmount);
        return err;
    }
    function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec// Error checking
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
            return (fail(Error(error), FailureInfo.REPAY_BORROW_ACCRUE_INTEREST_FAILED), 0);
        }
        // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
        return repayBorrowFresh(msg.sender, msg.sender, repayAmount);
    }
    //knownsec//repayBorrowFresh Logical design
    function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns (uint, uint) {
        /* Fail if repayBorrow not allowed */
        uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower,
```

```
repayAmount);
        if (allowed != 0) {
            return (
            failOpaque(
                Error.COMPTROLLER_REJECTION,
                FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION,
                allowed
            ), 0);
        }
        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return                                          (fail(Error.MARKET_NOT_FRESH,
FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0);
        }
        RepayBorrowLocalVars memory vars;
        /* We remember the original borrowerIndex for verification purposes */
        vars.borrowerIndex = accountBorrows[borrower].interestIndex;

        /* We fetch the amount the borrower owes, with accumulated interest */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(
                Error.MATH_ERROR,

FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
,
                uint(vars.mathErr)
            ), 0);
        }

        /* If repayAmount == -1, repayAmount = accountBorrows */
        if (repayAmount == uint(- 1)) {
            vars.repayAmount = vars.accountBorrows;
```

```
        } else {
            vars.repayAmount = repayAmount;
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call doTransferIn for the payer and the repayAmount
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken holds an additional repayAmount of cash.
         *  doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
         *  it returns the amount actually transferred, in case of a fee.
         */
        vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow:
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.actualRepayAmount);
        require(vars.mathErr == MathError.NO_ERROR, "REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

        (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.actualRepayAmount);
        require(vars.mathErr == MathError.NO_ERROR,
```

```
"REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a RepayBorrow event */
        emit RepayBorrow(payer, borrower, vars.actualRepayAmount,
vars.accountBorrowsNew, vars.totalBorrowsNew);
        /* We call the defense hook */
        comptroller.repayBorrowVerify(address(this), payer, borrower,
vars.actualRepayAmount, vars.borrowerIndex);
        return (uint(Error.NO_ERROR), vars.actualRepayAmount);
    }
```

Safety advice: none.

3.21. repayBorrowBehalf logic design[Pass]

Audit half of the logic returned by repayBorrowBehalf to check whether the parameters are checked and the logic design is reasonable.

Audit results: No related security risks were found.

```
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint)
{
        (uint err,) = repayBorrowBehalfInternal(borrower, repayAmount);
        return err;
}
    function repayBorrowBehalfInternal(address borrower, uint repayAmount) internal
nonReentrant returns (uint, uint) {
        uint error = accrueInterest();
```

```
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
                return (fail(Error(error), FailureInfo.REPAY_BEHALF_ACCRUE_INTEREST_FAILED), 0);
            }
            // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
            return repayBorrowFresh(msg.sender, borrower, repayAmount);
        }
```

Safety advice: none.

3.22. liquidateBorrow logic design[Pass]

Audit the liquidateBorrow logic design, check the rationality of the logic design and check the parameters.

Audit results: No related security risks were found.

```
    function liquidateBorrow(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) external returns (uint) {
        (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
        return err;
    }
    //knownsec//liquidateBorrowInternal Logical
    function liquidateBorrowInternal(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) internal nonReentrant returns (uint, uint) {
```

```
            uint error = accrueInterest();

            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but we still want to log the fact that an attempted liquidation failed
                return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED), 0);
            }

            error = cTokenCollateral.accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but we still want to log the fact that an attempted liquidation failed
                return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED), 0);
            }
            // liquidateBorrowFresh emits borrow-specific logs on errors, so we don't need to
            return liquidateBorrowFresh(msg.sender, borrower, repayAmount, cTokenCollateral);
        }
        //knownsec//liquidateBorrowFresh Logical
        function liquidateBorrowFresh(
            address liquidator,
            address borrower,
            uint repayAmount,
            CTokenInterface cTokenCollateral
        ) internal returns (uint, uint) {
            /* Fail if liquidate not allowed */
            uint allowed = comptroller.liquidateBorrowAllowed(
                address(this),
                address(cTokenCollateral),
                liquidator,
                borrower,
```

```
                repayAmount
            );
            if (allowed != 0) {
                return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
            }
            /* Verify market's block number equals current block number */
            if (accrualBlockNumber != getBlockNumber()) {
                return (fail(Error.MARKET_NOT_FRESH,
FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
            }

            /* Verify cTokenCollateral market's block number equals current block number */
            if (cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
                return (fail(Error.MARKET_NOT_FRESH,
FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK), 0);
            }

            /* Fail if borrower = liquidator */
            if (borrower == liquidator) {
                return (fail(Error.INVALID_ACCOUNT_PAIR,
FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER), 0);
            }

            /* Fail if repayAmount = 0 */
            if (repayAmount == 0) {
                return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED,
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_ZERO), 0);
            }

            /* Fail if repayAmount = -1 */
            if (repayAmount == uint(- 1)) {
                return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED,
```

```
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
        }

        /* Fail if repayBorrow fails */
        (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator,
borrower, repayAmount);
        if (repayBorrowError != uint(Error.NO_ERROR)) {
            return (fail(Error(repayBorrowError),
FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED), 0);
        }

        //////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We calculate the number of collateral tokens that will be seized */
        (uint amountSeizeError, uint seizeTokens) =
comptroller.liquidateCalculateSeizeTokens(
            address(this),
            address(cTokenCollateral),
            actualRepayAmount
        );
        require(amountSeizeError == uint(Error.NO_ERROR),
"LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");
        /* Revert if borrower collateral token balance < seizeTokens */
        require(cTokenCollateral.balanceOf(borrower) >= seizeTokens,
"LIQUIDATE_SEIZE_TOO_MUCH");
        // If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make
an external call
        uint seizeError;
        if (address(cTokenCollateral) == address(this)) {
            seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
        } else {
```

```
            seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
        }
        /* Revert if seize tokens fails (since we cannot be sure of side effects) */
        require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

        /* We emit a LiquidateBorrow event */
        emit LiquidateBorrow(liquidator, borrower, actualRepayAmount, address(cTokenCollateral), seizeTokens);

        /* We call the defense hook */
        comptroller.liquidateBorrowVerify(
            address(this),
            address(cTokenCollateral),
            liquidator,
            borrower,
            actualRepayAmount,
            seizeTokens
        );

        return (uint(Error.NO_ERROR), actualRepayAmount);
    }
```

Safety advice: none.

3.23. _addReserves logic design[Pass]

Audit the _addReserves increase reserve logic to check whether the parameters are verified and the logic design is reasonable.

Audit results: No related security risks were found.

```
    function _addReserves(uint addAmount) external returns (uint) {
        return _addReservesInternal(addAmount);
    }
```

```
    function _addReservesInternal(uint addAmount) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {//knownsec// Error type checking
            // accrueInterest emits logs on errors,
            // but on top of that we want to log the fact that an attempted reduce reserves failed.
            return fail(Error(error), FailureInfo.ADD_RESERVES_ACCRUE_INTEREST_FAILED);
        }
        // _addReservesFresh emits reserve-addition-specific logs on errors, so we don't need to.
        (error,) = _addReservesFresh(addAmount);//knownsec//transfer addReservesFresh
        return error;
    }
    //knownsec//addReservesFresh logic
    function _addReservesFresh(uint addAmount) internal returns (uint, uint) {
        // totalReserves + actualAddAmount
        uint totalReservesNew;
        uint actualAddAmount;

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.ADD_RESERVES_FRESH_CHECK), actualAddAmount);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call doTransferIn for the caller and the addAmount
         *  Note: The cToken must handle variations between ERC-20 and ETH
```

```
underlying.
     *  On success, the cToken holds an additional addAmount of cash.
     *  doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
     *  it returns the amount actually transferred, in case of a fee.
     */

    actualAddAmount = doTransferIn(msg.sender, addAmount);

    totalReservesNew = totalReserves + actualAddAmount;

    /* Revert on overflow */
    require(totalReservesNew >= totalReserves, "add reserves unexpected overflow");

    // Store reserves[n+1] = reserves[n] + actualAddAmount
    totalReserves = totalReservesNew;

    /* Emit NewReserves(admin, actualAddAmount, reserves[n+1]) */
    emit ReservesAdded(msg.sender, actualAddAmount, totalReservesNew);

    /* Return (NO_ERROR, actualAddAmount) */
    return (uint(Error.NO_ERROR), actualAddAmount);
}
```

Safety advice: none.

3.24. _reduceReserves logic design[Pass]

Audit the _reduceReserves reserve reduction logic, check the rationality of the logic design, and whether the permission design and inspection are reasonable.

Audit results: No related security risks were found.

```
    function _reduceReserves(uint reduceAmount) external nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors,
            // but on top of that we want to log the fact that an attempted reduce reserves failed.
            return fail(Error(error), FailureInfo.REDUCE_RESERVES_ACCRUE_INTEREST_FAILED);
        }
        // _reduceReservesFresh emits reserve-reduction-specific logs on errors, so we don't need to.
        return _reduceReservesFresh(reduceAmount);
    }
    function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
        // totalReserves - reduceAmount
        uint totalReservesNew;

        // Check caller is admin
        if (msg.sender != admin) {//knownsec// Permission check
            return fail(Error.UNAUTHORIZED, FailureInfo.REDUCE_RESERVES_ADMIN_CHECK);
        }

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDUCE_RESERVES_FRESH_CHECK);
        }

        // Fail gracefully if protocol has insufficient underlying cash
        if (getCashPrior() < reduceAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.REDUCE_RESERVES_CASH_NOT_AVAILABLE);
```

```
        }

        // Check reduceAmount ≤ reserves[n] (totalReserves)
        if (reduceAmount > totalReserves) {
            return fail(Error.BAD_INPUT, FailureInfo.REDUCE_RESERVES_VALIDATION);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        totalReservesNew = totalReserves - reduceAmount;
        // We checked reduceAmount <= totalReserves above, so this should never revert.
        require(totalReservesNew <= totalReserves, "reduce reserves unexpected underflow");

        // Store reserves[n+1] = reserves[n] - reduceAmount
        totalReserves = totalReservesNew;

        // doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
        doTransferOut(feeManager, reduceAmount);
        emit ReservesReduced(admin, reduceAmount, totalReservesNew);
        return uint(Error.NO_ERROR);
    }
```

Safety advice: none.

3.25. isContractCall logic design[Pass]

Audit the isContractCall contract detection logic to check the rationality of the logic design.

Audit results: No related security risks were found.

```
 function isContractCall() public view returns (bool isContract){
      return msg.sender != tx.origin;
}
```

Safety advice: none.

4. Basic code vulnerability detection

4.1. Reentry attack detection [Pass]

Re-entry vulnerabilities are the most famous smart contract vulnerabilities.

In Solidity, calling functions of other addresses or transferring money to the contract address will pass your own address as the msg.sender of the called contract. At this time, if the energy passed is enough, it may be reentrant attacked by the other party. Invoking transfer/sender to the contract address in TRON will only pass 2300 Energy, which is not enough to launch a reentry attack. Therefore, you must avoid calling unknown contract addresses through call.value. Because call.value can pass in energy much larger than 2300.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.2. Replay attack detection [Pass]

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts. .

Detection result: After detection, the smart contract does not use the call function,

and this vulnerability does not exist.

Safety advice: None.

4.3. Detection of rearrangement attacks [Pass]

A rearrangement attack is when miners or other parties try to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Safety advice: None.

4.4. Numerical overflow detection [Pass]

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.5. Arithmetic accuracy error [Pass]

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages—Solidity does not float. Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same-level calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.6. Access control detection [Pass]

Different functions in the contract should set reasonable permissions

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly

defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.7. tx.origin authentication [Pass]

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Inappropriate use of this variable for authentication in smart contracts can make the contract vulnerable to attacks like phishing.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.8. call injection attack [Pass]

When calling the call function, strict permission control should be done, or the function called by the call should be written dead.

Detection result: After detection, the smart contract does not use the call function, and this vulnerability does not exist.

Safety advice: None.

4.9. Return value call verification [Pass]

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

There are currency transfer methods such as transfer(), send(), call.value() in Solidity, which can all be used to send trx to a certain address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300energy will be transferred for calling to prevent reentry attacks; false will be returned when send fails; only 2300energy will be transferred for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available energy will be transferred for calling (can be By passing in feeLimit parameters to limit), reentry attacks cannot be effectively prevented.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may cause unexpected results due to trx sending failure.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.10. Uninitialized storage pointer [Pass]

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Test result: After testing, the smart contract code does not use structure, and there is no such problem.

Safety advice: None.

4.11. Wrong use of random numbers [Pass]

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access obviously unpredictable values, such as block.number and block.timestamp, they are usually either more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.12. Transaction order depends on [Pass]

Since miners always obtain energy fees through codes that represent externally

owned addresses (EOA), users can specify higher fees for faster transactions. Since the TRON blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.13. Denial of Service Attack [Pass]

In the world of TRON, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the energy required for computing functions to cause energy depletion, abusing access control to access the private components of the smart contract, using confusion and negligence, etc. Wait.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.14. Fake recharge loophole [Pass]

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, it enters the else logic part and returns false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in the scene of sensitive functions such as transfer.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.15. Vulnerabilities in the issuance of additional tokens[Pass]

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

Test result: After testing, there is a token issuance logic in the smart contract code, but it belongs to the normal business needs of the liquidity pool minting/destroying logic and has a clear permission check, and it is assessed as passed.

Safety advice: None.

4.16. Bypassing frozen account[Pass]

Check whether the token source account, the originating account, and the target

account are frozen when transferring tokens in the token contract.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.17. Compiler version security [Pass]

Check whether a safe compiler version is used in the contract code implementation

Test result: After testing, the smart contract code has a compiler version 0.5.8 or higher, and there is no such security issue.

Safety advice: None.

4.18. Unrecommended encoding method [Pass]

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.19. Redundant code [Pass]

Check whether the contract code implementation contains redundant code

Test result: After testing, the security problem does not exist in the smart contract

code.

Safety advice: None.

4.20. The use of safe arithmetic library [Pass]

Check whether the SafeMath safe arithmetic library is used in the contract code implementation

Test result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Safety advice: None.

4.21. Use of require/assert [Pass]

Check the rationality of the use of require and assert statements in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.22. Energy consumption test [Pass]

Check whether the energy consumption exceeds the maximum block limit

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.23. Fallback function safety [Pass]

Check whether the fallback function is used correctly in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.24. Owner permission control [Pass]

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.25. Security of low-level functions [Pass]

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.26. Variable coverage [Pass]

Check whether there are security issues caused by variable coverage in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.27. Timestamp dependency attack [Pass]

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it is only necessary to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that rely on timestamps in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4.28. Unsafe interface usage [Pass]

Check whether unsafe interfaces are used in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

5. Appendix A: Contract code

Source code for this test:

FeeManager.sol

```
pragma solidity ^0.5.9;

import "../token/RewardToken.sol";
import "../lib/Controller.sol";
import "../interface/IBool.sol";
import "../interface/IFactory.sol";
import "../interface/Converter.sol";
import "../lib/SafeMath.sol";

contract FeeManager is Controller {

    using SafeMath for uint;
    uint public constant MAX = 2 ** 256 - 1;
    uint public constant MIN_FEE = 1e12;
    uint public constant MAX_FEE = 1e18;

    uint public goverFundDivRate = 3 * 1e16;
    address public governaddr;

    uint public burnRate = 15 * 1e16;

    uint public dexTokenAmount = 10000 * 1e18;

    address public boxToken;
    address public baseToken;
    address public basePool;

    address public factory;

    uint private setFactoryManagerCount = 0;
    uint private setBasePoolCount = 0;
    address payable public wtrxAddress;

    mapping(address => address) public converters;


    constructor() Controller() public {
    }

    function() payable external {}


    function convertWtrx() external {
        uint _amount = address(this).balance;
        wtrxAddress.transfer(_amount);
    }

    function setWtrxAddress(address payable _address) onlyController external {
        wtrxAddress = _address;
    }


    function setFactory(address _factory) onlyController external {
        require(setFactoryManagerCount < 2, "limit factory address");
        factory = _factory;
        setFactoryManagerCount = setFactoryManagerCount.add(1);
    }

    function setDexTokenAmount(uint _amount) onlyController external {
        dexTokenAmount = _amount;
    }

    function swapToken(
        address poolAddress, address tokenIn, address tokenOut
    ) onlyController public returns (uint){
        require(factory != address(0), "factory=0");
        require(IFactory(factory).isBPool(poolAddress), "!pool");
        uint _tokenInAmount = IERC20(tokenIn).balanceOf(address(this));
        IERC20(tokenIn).approve(poolAddress, _tokenInAmount);
        (uint tokenAmountOut,) = IBool(poolAddress).swapExactAmountIn(tokenIn, _tokenInAmount,
tokenOut, 0, MAX);
        return tokenAmountOut;
    }

    function addTokenPoolLiquidity() external {
        require(IERC20(boxToken).balanceOf(msg.sender) >= dexTokenAmount, "not enough
token");
```

```
        uint _amount = IERC20(baseToken).balanceOf(address(this));
        uint256 _goverFund = _amount.mul(goverFundDivRate).div(1e18);
        IERC20(baseToken).transfer(governaddr, _goverFund);

        uint _tokenInAmount = _amount.mul(burnRate).div(1e18);

        IERC20(baseToken).approve(basePool, _tokenInAmount);
        IBool(basePool).swapExactAmountIn(baseToken, _tokenInAmount, boxToken, 0, MAX);

        _amount = _amount.sub(_goverFund);
        _amount = _amount.sub(_tokenInAmount);

        IERC20(baseToken).approve(basePool, _amount);
        IBool(basePool).joinswapExternAmountIn(baseToken, _amount, 0);
    }


    function burnToken() public {
        uint _amount = IERC20(boxToken).balanceOf(address(this));
        if (_amount > 0) {
            RewardToken(boxToken).burn(_amount);
        }
    }


    function setBasePoolToken(address _pool, address _token, address _baseToken) onlyController
public {
        require(factory != address(0), "factory is 0");
        require(IFactory(factory).isBPool(_pool), "!pool");
        require(IBool(_pool).isBound(_token), "not bound");
        require(IBool(_pool).isBound(_baseToken), "not bound");
        require(setBasePoolCount < 2, "limit set base pool");
        setBasePoolCount = setBasePoolCount.add(1);
        basePool = _pool;
        boxToken = _token;
        baseToken = _baseToken;

    }

    function gover(address _goveraddr) onlyController public {
        governaddr = _goveraddr;
    }

    function setBurnRate(uint _rate) onlyController public {
        require(_rate < 5 * 1e17, "< MAX_FEE");
        burnRate = _rate;
    }

    function setGoverFundDivRate(uint256 _goverFundDivRate) onlyController public {
        require(_goverFundDivRate < MAX_FEE, "< MAX_FEE");
        goverFundDivRate = _goverFundDivRate;
    }

    function convert(address _token, address _to) onlyController public {
        address converter = converters[_token];
        Converter(converter).convert(_to);
    }

    function setConverter(
        address _token,
        address _converter
    ) public onlyController {
        converters[_token] = _converter;
    }

}
```

AssertPriceOracle.sol

```
pragma solidity ^0.5.9;

import "../lib/Controller.sol";
import "../interface/PriceOracle.sol";
import "../interface/ILinkOracle.sol";
import "../interface/IJustPair.sol";
import "../interface/IPoolView.sol";
import "../CErc20.sol";
import "../lib/SafeMath.sol";

contract AssertPriceOracle is PriceOracle, Controller {
    using SafeMath for uint;

    //underlying address => oracle address
    mapping(address => address) public underlyingOracles;
    //pair address => underlying address
    mapping(address => address) public justLpPairs;
```

```
    mapping(address => bool) public justLps;

    // lp => pool view
    mapping(address => address) public abeloLpViews;

    mapping(address => bool) public abeloLps;

    address public wtrxAddress;
    uint public wtrxDecimals = 6;

    constructor(address _controller, address _wtrxAddress) Controller() public {
        setController(_controller);
        wtrxAddress = _wtrxAddress;
    }


    function toUint(bytes32 inBytes) pure public returns (uint256 outUint) {
        return uint256(inBytes);
    }

    function setUnderlyingOracle(address underlying, address oracle) onlyController external {
        ILinkOracle(oracle).latestAnswer();
        underlyingOracles[underlying] = oracle;
    }


    function getUnderlyingPrice(CToken cToken) external view returns (uint){
        address _underlying = address(CErc20(address(cToken)).underlying());
        if (justLps[_underlying] == true) {
            return calJustLpPrice(_underlying);
        }
        if (abeloLps[_underlying] == true) {
            return calAbeloLpPrice(_underlying);
        }
        address oracle = underlyingOracles[_underlying];
        require(oracle != address(0), "oracle is 0");
        return toUint(ILinkOracle(oracle).latestAnswer());

    }

    function addAbeloPairView(address _lpAddress, address _poolView) onlyController external {
        //            require(IPoolView(_poolView).pool() != _lpAddress, "pool is error");
        abeloLps[_lpAddress] = true;
        abeloLpViews[_lpAddress] = _poolView;


    }

    function addJustLp(address _underlying, address _pairaddress) onlyController external {
        require(_underlying == IJustPair(_pairaddress).tokenAddress(), "pair is error");
        justLps[_pairaddress] = true;
        justLpPairs[_pairaddress] = _underlying;
    }

    function calJustLpPrice(address _pairAddress) public view returns (uint){
        address _underlying = justLpPairs[_pairAddress];

        require(underlyingOracles[_underlying] != address(0), "oracle is 0");

        uint _totalSupply = IJustPair(_pairAddress).totalSupply().div(10 **
IJustPair(_pairAddress).decimals());

        uint _totalUnderlying = ERC20Detailed(_underlying).balanceOf(_pairAddress);
        uint _totalTrx = _pairAddress.balance;
        uint _underlyingDecimal = ERC20Detailed(_underlying).decimals();

        uint lpPerUnderlying = _totalUnderlying.div(_totalSupply).div(10 **
_underlyingDecimal);

        uint _lpPerTrx = _totalTrx.div(_totalSupply).div(10 ** wtrxDecimals);
        address oracle = underlyingOracles[_underlying];

        uint _underLyingPrice = uint(ILinkOracle(oracle).latestAnswer());

        address wtrxOracle = underlyingOracles[wtrxAddress];

        uint _trxPrice = uint(ILinkOracle(wtrxOracle).latestAnswer());

        return _lpPerUnderlying.mul(_underLyingPrice).add(_lpPerTrx.mul(_trxPrice));

    }

    function calAbeloLpPrice(address _lpAddress) public view returns (uint){

        address _poolView = abeloLpViews[_lpAddress];
        uint total;
        address[] memory tokens = IPoolView(_poolView).getCurrentTokens();
```

```
            for (uint i = 0; i < 2; i++) {
                address token = tokens[i];
                uint _lpPerToken = IPoolView(_poolView).getTokenAmountPerLp(token);
                address oracle = underlyingOracles[token];
                uint _tokenPrice = uint(ILinkOracle(oracle).latestAnswer());
                total = total.add(_lpPerToken.mul(_tokenPrice));
            }
            return total;
        }

}
```

SimplePriceOracle.sol

```
pragma solidity ^0.5.9;

import "../interface/PriceOracle.sol";
import "../CErc20.sol";

contract SimplePriceOracle is PriceOracle {
    mapping(address => uint) prices;

    event PricePosted(address asset, uint previousPriceMantissa, uint requestedPriceMantissa, uint
newPriceMantissa);

    function getUnderlyingPrice(CToken cToken) public view returns (uint) {
        if (compareStrings(cToken.symbol(), "cETH")) {
            return 1e18;
        } else {
            return prices[address(CErc20(address(cToken)).underlying())];
        }
    }

    function setUnderlyingPrice(CToken cToken, uint underlyingPriceMantissa) public {
        address asset = address(CErc20(address(cToken)).underlying());
        emit PricePosted(asset, prices[asset], underlyingPriceMantissa, underlyingPriceMantissa);
        prices[asset] = underlyingPriceMantissa;
    }

    function setDirectPrice(address asset, uint price) public {
        emit PricePosted(asset, prices[asset], price, price);
        prices[asset] = price;
    }

    // v1 price oracle interface for use as backing of proxy
    function assetPrices(address asset) external view returns (uint) {
        return prices[asset];
    }

    function compareStrings(string memory a, string memory b) internal pure returns (bool) {
        return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
    }
}
```

IRewardPool.sol

```
pragma solidity ^0.5.9;

interface IRewardPool {
    function notifyRewardAmount(uint256 reward) external;
}
```

LPTokenWrapper.sol

```
pragma solidity ^0.5.0;

import "../lib/SafeMath.sol";
import "../lib/ERC20.sol";

contract LPTokenWrapper {
    using SafeMath for uint256;


    IERC20 public lpToken;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;

    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
```

```
    }

    function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        lpToken.transferFrom(msg.sender, address(this), amount);
    }

    function withdraw(uint256 amount) public {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        lpToken.transfer(msg.sender, amount);
    }

    function migrateStakeFor(address target, uint256 amountNewShare) internal    {
        _totalSupply = _totalSupply.add(amountNewShare);
        _balances[target] = _balances[target].add(amountNewShare);
    }
}
```

Reservoir.sol

```
pragma solidity ^0.5.9;


import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Controller.sol";
import "../lib/SafeMath.sol";
import "../lib/IERC20.sol";
import "../lib/EnumerableSet.sol";
import "./RewardToken.sol";
import "./IRewardPool.sol";
import "../Comptroller.sol";

contract Reservoir is Controller {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;

    Comptroller public comptroller;

    uint256 poolLastRewardBlock;

    uint256 fundLastRewardBlock;

    struct PoolInfo {
        uint256 allocPoint; // How many allocation points assigned to this pool. dex to distribute per
block.
        address contractAddress;
    }


    // 30 days
    uint public duration = 30 * 28800;

    RewardToken public rewardToken;
    // Dev fund (25%, initially)
    uint public teamFundDivRate = 25e16;
    // gover fund (5%, initially)
    uint public goverFundDivRate = 5e16;
    // insurance fund (5%, initially)
    uint public insuranceFundDivRate = 5e16;
    uint public liquidityRate = 3e16;
    uint public comptrollerRate = 62e16;
    uint256 public MAX_RATE = 1e18;

    uint256 public tokenPerBlock = 5 * 1e18;
    // tokens created per block.
    uint256 public tokenPerBlockForReward;
    uint256 public MIN_TOKEN_REWARD = 5 * 1e16;


    EnumerableSet.AddressSet private teamAddrs;

    address public governaddr;
    address public insuranceaddr;


    PoolInfo[] public poolInfo;

    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    uint256 decrement = 0;
    // The block number when PICKLE mining starts.
    uint256 public startBlock;
```

```
uint256 public periodEndBlock;

// Events
event Recovered(address token, uint256 amount);

constructor(
        RewardToken _rewardToken,
        uint256 _startBlock,
        uint256 _duration
) Controller() public {
        periodEndBlock = _startBlock.add(_duration);
        require(periodEndBlock > block.number, "end is wrong");
        rewardToken = _rewardToken;
        startBlock = _startBlock;
        fundLastRewardBlock = _startBlock;
        duration = _duration;
        tokenPerBlockForReward = calTokenPerBlock(tokenPerBlock);

}


function calTokenPerBlock(uint256 _blockToken) view internal returns (uint256){
        uint256 _devFund = calRate(_blockToken, teamFundDivRate);
        uint256 _goverFund = calRate(_blockToken, goverFundDivRate);
        uint256 _insuranceFund = calRate(_blockToken, insuranceFundDivRate);
        return _blockToken.sub(_devFund).sub(_goverFund).sub(_insuranceFund);
}

function mintToken() onlyOwner public {
        rewardToken.mint(address(this));
}

function poolLength() external view returns (uint256) {
        return poolInfo.length;
}

// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(
        address _contractAddress,
        uint256 _allocPoint
) public onlyController adjustProduct {
        uint256 lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
                PoolInfo(
                {
                allocPoint : _allocPoint,
                contractAddress : _contractAddress
                }
                )
        );
}

function set(
        uint256 _pid,
        uint256 _allocPoint
) public onlyController adjustProduct {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
                _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
}

modifier adjustProduct()    {
        .
        if (block.number >= periodEndBlock) {
                if (tokenPerBlock > MIN_TOKEN_REWARD) {
                        tokenPerBlock = tokenPerBlock.mul(90).div(100);
                }
                if (tokenPerBlock < MIN_TOKEN_REWARD) {
                        tokenPerBlock = MIN_TOKEN_REWARD;
                }

                periodEndBlock = block.number.add(duration);
        }
}
function calRate(uint _amount, uint rate) public view returns (uint){
        return _amount.mul(rate).div(1e18);
}

// Distribute the output of a cycle in advance
function distributionToken() adjustProduct public {
        if (block.number < startBlock) {
                return;
```

```
        }
        if (poolLastRewardBlock >= periodEndBlock) {
            return;
        }
        uint multiplier = 1;
        if (periodEndBlock > block.number) {
            multiplier = periodEndBlock - block.number;
        }
        uint _reward = multiplier.mul(tokenPerBlock);

        uint goverFund = calRate(_reward, goverFundDivRate);
        uint insuranceFund = calRate(_reward, insuranceFundDivRate);
        uint teamFund = calRate(_reward, teamFundDivRate);
        uint liquidityFund = calRate(_reward, liquidityRate);

        uint distriReward =
_reward.sub(goverFund.add(insuranceFund).add(teamFund).add(liquidityFund));

        safeTokenTransfer(address(comptroller), distriReward);
        Comptroller(comptroller)._setCompRate(calRate(tokenPerBlock, comptrollerRate));

        uint length = poolInfo.length;
        for (uint pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            uint256 poolReward = liquidityFund.mul(pool.allocPoint).div(totalAllocPoint);
            safeTokenTransfer(pool.contractAddress, poolReward);
            IRewardPool(pool.contractAddress).notifyRewardAmount(poolReward);
        }
        poolLastRewardBlock = periodEndBlock;


    }

    // Safe pickle transfer function, just in case if rounding error causes pool to not have enough dex.
    function safeTokenTransfer(address _to, uint256 _amount) internal {
        uint256 pickleBal = rewardToken.balanceOf(address(this));
        if (_amount > pickleBal) {
            rewardToken.transfer(_to, pickleBal);
        } else {
            rewardToken.transfer(_to, _amount);
        }
    }

    // Distribute tokens according to teh speed of the block
    function claimFund() onlyController external {
        if (block.number <= fundLastRewardBlock) {
            return;
        }
        uint256 multiplier = block.number - fundLastRewardBlock;
        uint256 boxReward = multiplier.mul(tokenPerBlock);
        uint goverFund = calRate(boxReward, goverFundDivRate);
        uint insuranceFund = calRate(boxReward, insuranceFundDivRate);
        uint teamFund = calRate(boxReward, teamFundDivRate);
        rewardToken.transfer(governaddr, goverFund);
        rewardToken.transfer(insuranceaddr, insuranceFund);
        uint256 perDevFund = teamFund.div(teamAddrs.length());
        for (uint256 i = 0; i < teamAddrs.length() - 1; i++) {
            rewardToken.transfer(teamAddrs.get(i), perDevFund);
        }
        uint256 remainFund = teamFund.sub(perDevFund.mul(teamAddrs.length() - 1));
        rewardToken.transfer(teamAddrs.get(teamAddrs.length() - 1), remainFund);
        goverFund = 0;
        insuranceFund = 0;
        teamFund = 0;
        fundLastRewardBlock = block.number;
        distributionToken();

    }


    function setComptroller(address _address) onlyController external {
        comptroller = Comptroller(_address);
    }

    function addDev(address _devaddr) onlyController public {
        require(teamAddrs.length() < 50, "less 50");
        teamAddrs.add(_devaddr);
    }

    function removeDev(address _devaddr) onlyController public {
        teamAddrs.remove(_devaddr);
    }

    function contains(address _dev) public view returns (bool){
        return teamAddrs.contains(_dev);
    }
```

```
    function length() public view returns (uint256){
        return teamAddrs.length();
    }

    function setTeamFundDivRate(uint256 _teamFundDivRate) onlyController public {
        require(_teamFundDivRate <= 25e16, "rate too large");
        teamFundDivRate = _teamFundDivRate;
        comptrollerRate
        =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
    }

    function setLiquidityRate(uint _rate) onlyController external {
        require(_rate <= 10e16, "rate too large");
        liquidityRate = _rate;
        comptrollerRate =

MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
    }

    function insurance(address _insuranceaddr) onlyController public {
        insuranceaddr = _insuranceaddr;
    }

    function setInsuranceFundDivRate(uint256 _insuranceFundDivRate) onlyController public {
        require(_insuranceFundDivRate <= 5e16, "rate too large");
        insuranceFundDivRate = _insuranceFundDivRate;
        comptrollerRate
        =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
    }

    function gover(address _goveraddr) onlyController public {
        governaddr = _goveraddr;
    }

    function setGoverFundDivRate(uint256 _goverFundDivRate) onlyController public {
        require(_goverFundDivRate <= 5e16, "rate too large");
        goverFundDivRate = _goverFundDivRate;
        comptrollerRate
        =
MAX_RATE.sub(teamFundDivRate).sub(goverFundDivRate).sub(insuranceFundDivRate).sub(liquidity
Rate);
    }

}
```

RewardPool.sol

```
pragma solidity ^0.5.9;

import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Controller.sol";
import "../lib/Math.sol";
import "../lib/SafeMath.sol";
import "../lib/IERC20.sol";
import "../lib/EnumerableSet.sol";
import "./RewardToken.sol";
import "./LPTokenWrapper.sol";
import "../lib/Controller.sol";


contract RewardPool is LPTokenWrapper, Controller {

    IERC20 public rewardToken;
    uint256 public duration; // making it not a constant is less gas efficient, but portable

    uint256 public periodFinish = 0;
    uint256 public rewardRate = 0;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;

    uint256 public exitFee = 3e16;
    address public feeManager;

    address public reservoirAddress;

    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    mapping(address => bool) smartContractStakers;
```

```
    event RewardAdded(uint256 reward);
    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event RewardPaid(address indexed user, uint256 reward);
    event RewardDenied(address indexed user, uint256 reward);
    event SmartContractRecorded(address indexed smartContractAddress, address indexed smartContractInitiator);

    // Harvest Migration
    event Migrated(address indexed account, uint256 legacyShare, uint256 newShare);

    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }

    constructor(
        address _reservoirAddress,
        address _rewardToken,
        address _lpToken,
        uint256 _duration,
        address _controller,
        address _feeManager
    ) public
    Controller(){
        reservoirAddress = _reservoirAddress;
        rewardToken = IERC20(_rewardToken);
        lpToken = IERC20(_lpToken);
        feeManager = _feeManager;
        duration = _duration;
        setController(_controller);
    }

    function setExitFee(uint _fee) public onlyController {
        require(_fee <= 1e17, "fee too much");
        exitFee = _fee;
    }

    function lastTimeRewardApplicable() public view returns (uint256) {
        return Math.min(block.number, periodFinish);
    }

    function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
        rewardPerTokenStored.add(
            lastTimeRewardApplicable()
            .sub(lastUpdateTime)
            .mul(rewardRate)
            .mul(1e18)
            .div(totalSupply())
        );
    }

    function earned(address account) public view returns (uint256) {
        return
        balanceOf(account)
        .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
        .div(1e18)
        .add(rewards[account]);
    }

    // stake visibility is public as overriding LPTokenWrapper's stake() function
    function stake(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot stake 0");
        super.stake(amount);
        emit Staked(msg.sender, amount);
    }

    function withdraw(uint256 amount) public updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");
        uint exitAmount = amount.mul(exitFee).div(1e18);
        uint _amount = amount.sub(exitAmount);
        super.withdraw(_amount);
        emit Withdrawn(msg.sender, amount);
    }

    function exit() external {
        withdraw(balanceOf(msg.sender));
        getReward();
```

```
    }

    function pushReward(address recipient) public updateReward(recipient) onlyController {
        uint256 reward = earned(recipient);
        if (reward > 0) {
            rewards[recipient] = 0;
            rewardToken.transfer(recipient, reward);
            emit RewardPaid(recipient, reward);

        }
    }

    function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
            rewards[msg.sender] = 0;
            // If it is a normal user and not smart contract,
            // then the requirement will pass
            // If it is a smart contract, then
            // make sure that it is not on our greyList.
            if (tx.origin == msg.sender) {
                rewardToken.transfer(msg.sender, reward);
                emit RewardPaid(msg.sender, reward);
            } else {
                emit RewardDenied(msg.sender, reward);
            }
        }
    }

    function notifyRewardAmount(uint256 reward)
    external
    updateReward(address(0))
    {
        require(msg.sender == controller() || msg.sender == reservoirAddress, "only controller and reservoir");

        require(reward < uint(- 1) / 1e18, "the notified reward cannot invoke multiplication overflow");

        if (block.number >= periodFinish) {
            rewardRate = reward.div(duration);
        } else {
            uint256 remaining = periodFinish.sub(block.number);
            uint256 leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(duration);
        }
        lastUpdateTime = block.number;
        periodFinish = block.number.add(duration);
        emit RewardAdded(reward);
    }

    function setReservoir(address _reservoir) onlyController public {
        require(_reservoir != address(0), "reservoir is not 0");
        reservoirAddress = _reservoir;

    }
}
```

RewardToken,.sol

```
pragma solidity ^0.5.9;
pragma experimental ABIEncoderV2;

import "../lib/ERC20.sol";
import "../lib/ERC20Detailed.sol";
import "../lib/Ownable.sol";
import "../lib/ERC20Burnable.sol";

contract RewardToken is ERC20, ERC20Detailed, ERC20Burnable, Ownable {

    uint public constant _totalSupply = 43200000e18;


    constructor () public ERC20Detailed("BOX-BANK", "BOX", 18) {
    }

    function mint(address _to) public onlyOwner {
        require(totalSupply() == 0, "mint once");
        _mint(_to, _totalSupply);
    }

}
```

BaseJumpRateModelV2.sol

```
pragma solidity ^0.5.9;

import "./lib/SafeMath.sol";

/**
  * @title Logic for Compound's JumpRateModel Contract V2.
  * @author Compound (modified by Dharma Labs, refactored by Arr00)
  * @notice Version 2 modifies Version 1 by enabling updateable parameters.
  */
contract BaseJumpRateModelV2 {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint jumpMultiplierPerBlock, uint kink);

    /**
     * @notice The address of the owner, i.e. the Timelock contract, which can update parameters directly
     */
    address public owner;

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 10512000;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
    uint public baseRatePerBlock;

    /**
     * @notice The multiplierPerBlock after hitting a specified utilization point
     */
    uint public jumpMultiplierPerBlock;

    /**
     * @notice The utilization point at which the jump multiplier is applied
     */
    uint public kink;

    /**
     * @notice Construct an interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     * @param owner_ The address of the owner,
     *               i.e. the Timelock contract (which has the ability to update parameters directly)
     */
    constructor(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_, address owner_
    ) internal {
        owner = owner_;

        updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
    }

    /**
     * @notice Update the parameters of the interest rate model (only callable by owner, i.e. Timelock)
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     */
    function updateJumpRateModel(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_
    ) external {
        require(msg.sender == owner, "only the owner may call this function.");

        updateJumpRateModelInternal(baseRatePerYear, multiplierPerYear, jumpMultiplierPerYear, kink_);
```

```
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getBorrowRateInternal(uint cash, uint borrows, uint reserves) internal view returns (uint) {
        uint util = utilizationRate(cash, borrows, reserves);

        if (util <= kink) {
            return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
        } else {
            uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            uint excessUtil = util.sub(kink);
            return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
        }
    }

    /**
     * @notice Calculates the current supply rate per block
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @param reserveFactorMantissa The current reserve factor for the market
     * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRateInternal(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }

    /**
     * @notice Internal function to update the parameters of the interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     */
    function updateJumpRateModelInternal(
        uint baseRatePerYear,
        uint multiplierPerYear,
        uint jumpMultiplierPerYear,
        uint kink_
    ) internal {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = (multiplierPerYear.mul(1e18)).div(blocksPerYear.mul(kink_));
        jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
        kink = kink_;

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
    }
}
```

CErc20.sol

```
pragma solidity ^0.5.9;
```

```
import "./CToken.sol";
import "./interface/CErc20Interface.sol";

/**
  * @title Compound's CErc20 Contract
  * @notice CTokens which wrap an EIP-20 underlying
  * @author Compound
  */
contract CErc20 is CToken, CErc20Interface {
      /**
        * @notice Initialize the new money market
        * @param underlying_ The address of the underlying asset
        * @param comptroller_ The address of the Comptroller
        * @param interestRateModel_ The address of the interest rate model
        * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
        * @param name_ ERC-20 name of this token
        * @param symbol_ ERC-20 symbol of this token
        * @param decimals_ ERC-20 decimal precision of this token
        */
      function initialize(address underlying_,
            ComptrollerInterface comptroller_,
            InterestRateModel interestRateModel_,
            uint initialExchangeRateMantissa_,
            string memory name_,
            string memory symbol_,
            uint8 decimals_) public {
            // CToken initialize does the bulk of the work
            super.initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_,
symbol_, decimals_);

            // Set underlying and sanity check it
            underlying = underlying_;
            IERC20(underlying).totalSupply();
      }

      /*** User Interface ***/

      /**
        * @notice Sender supplies assets into the market and receives cTokens in exchange
        * @dev Accrues interest whether or not the operation succeeds, unless reverted
        * @param mintAmount The amount of the underlying asset to supply
        * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
        */
      function mint(uint mintAmount) external returns (uint) {
            (uint err,) = mintInternal(mintAmount);
            return err;
      }

      /**
        * @notice Sender redeems cTokens in exchange for the underlying asset
        * @dev Accrues interest whether or not the operation succeeds, unless reverted
        * @param redeemTokens The number of cTokens to redeem into underlying
        * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
        */
      function redeem(uint redeemTokens) external returns (uint) {
            return redeemInternal(redeemTokens);
      }

      /**
        * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
        * @dev Accrues interest whether or not the operation succeeds, unless reverted
        * @param redeemAmount The amount of underlying to redeem
        * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
        */
      function redeemUnderlying(uint redeemAmount) external returns (uint) {
            return redeemUnderlyingInternal(redeemAmount);
      }

      /**
         * @notice Sender borrows assets from the protocol to their own address
         * @param borrowAmount The amount of the underlying asset to borrow
         * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
         */
      function borrow(uint borrowAmount) external returns (uint) {
            return borrowInternal(borrowAmount);
      }

      /**
        * @notice Sender repays their own borrow
        * @param repayAmount The amount to repay
        * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
        */
      function repayBorrow(uint repayAmount) external returns (uint) {
            (uint err,) = repayBorrowInternal(repayAmount);
            return err;
      }
```

```
    /**
      * @notice Sender repays a borrow belonging to borrower
      * @param borrower the account with the debt being payed off
      * @param repayAmount The amount to repay
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint) {
        (uint err,) = repayBorrowBehalfInternal(borrower, repayAmount);
        return err;
    }

    /**
      * @notice The sender liquidates the borrowers collateral.
      *  The collateral seized is transferred to the liquidator.
      * @param borrower The borrower of this cToken to be liquidated
      * @param repayAmount The amount of the underlying borrowed asset to repay
      * @param cTokenCollateral The market in which to seize collateral from the borrower
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function liquidateBorrow(
        address borrower,
        uint repayAmount,
        CTokenInterface cTokenCollateral
    ) external returns (uint) {
        (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
        return err;
    }

    /**
      * @notice The sender adds to reserves.
      * @param addAmount The amount fo underlying token to add as reserves
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _addReserves(uint addAmount) external returns (uint) {
        return _addReservesInternal(addAmount);
    }

    /*** Safe Token ***/

    /**
      * @notice Gets balance of this contract in terms of the underlying
      * @dev This excludes the value of the current message, if any
      * @return The quantity of underlying tokens owned by this contract
      */
    function getCashPrior() internal view returns (uint) {
        IERC20 token = IERC20(underlying);
        return token.balanceOf(address(this));
    }

    /**
      * @dev Similar to EIP20 transfer, except it handles a False result from `transferFrom` and reverts in that case.
      *      This will revert due to insufficient balance or insufficient allowance.
      *      This function returns the actual amount received,
      *      which may be less than `amount` if there is a fee attached to the transfer.
      *
      *      Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
      *            See here:
      https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
      */
    function doTransferIn(address from, uint amount) internal returns (uint) {
        EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
        uint balanceBefore = IERC20(underlying).balanceOf(address(this));
        token.transferFrom(from, address(this), amount);
        //todo EIP20Non
        //          bool success;
        //          assembly {
        //              switch returndatasize()
        //                  case 0 {// This is a non-standard ERC-20
        //                      success := not(0)          // set success to true
        //                  }
        //                  case 32 {// This is a compliant ERC-20
        //                      returndatacopy(0, 0, 32)
        //                      success := mload(0)        // Set `success = returndata` of external call
        //                  }
        //                  default {// This is an excessively non-compliant ERC-20, revert.
        //                      revert(0, 0)
        //                  }
        //          }
        //          require(success, "TOKEN_TRANSFER_IN_OVERFLOWSFER_IN_FAILED");

        // Calculate the amount that was *actually* transferred
        uint balanceAfter = IERC20(underlying).balanceOf(address(this));
        require(balanceAfter >= balanceBefore, "TOKEN_TRANSFER_IN_OVERFLOW");
```

```
            return balanceAfter - balanceBefore;
            // underflow already checked above, just subtract
        }

        /**
         * @dev Similar to EIP20 transfer, except it handles a False success from `transfer` and returns an explanatory
         *          error code rather than reverting.
         *          If caller has not called checked protocol's balance, this may revert due to
         *          insufficient cash held in this contract. If caller has checked protocol's balance prior to this call,
         *          and verified
         *          it is >= amount, this should not revert in normal conditions.
         *
         *          Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
         *                See here:
         *
         https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
         */
        //todo EIP20Non for tron usdt
        function doTransferOut(address payable to, uint amount) internal {
            EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
            token.transfer(to, amount);

            //              bool success;
            //              assembly {
            //                  switch returndatasize()
            //                  case 0 {// This is a non-standard ERC-20
            //                      success := not(0)          // set success to true
            //                  }
            //                  case 32 {// This is a complaint ERC-20
            //                      returndatacopy(0, 0, 32)
            //                      success := mload(0)        // Set `success = returndata` of external call
            //                  }
            //                  default {// This is an excessively non-compliant ERC-20, revert.
            //                      revert(0, 0)
            //                  }
            //              }
            //              require(success, "TOKEN_TRANSFER_OUT_FAILED");
        }
    }
```

CErc20Delegate.sol

```
pragma solidity ^0.5.9;

import "./CErc20.sol";
import "./interface/CDelegateInterface.sol";

/**
 * @title Compound's CErc20Delegate Contract
 * @notice CTokens which wrap an EIP-20 underlying and are delegated to
 * @author Compound
 */
contract CErc20Delegate is CErc20, CDelegateInterface {
    /**
     * @notice Construct an empty delegate
     */
    constructor() public {}

    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public {
        // Shh -- currently unused
        data;

        // Shh -- we don't ever want this hook to be marked pure
        if (false) {
            implementation = address(0);
        }

        require(msg.sender == admin, "only the admin may call _becomeImplementation");
    }

    /**
     * @notice Called by the delegator on a delegate to forfeit its responsibility
     */
    function _resignImplementation() public {
        // Shh -- we don't ever want this hook to be marked pure
        if (false) {
            implementation = address(0);
        }
```

```
            require(msg.sender == admin, "only the admin may call _resignImplementation");
        }
}
```

CErc20Delegator.sol

```
pragma solidity ^0.5.9;

import "./interface/CTokenInterface.sol";
import "./interface/CDelegatorInterface.sol";
import "./interface/CErc20Interface.sol";

/**
  * @title Compound's CErc20Delegator Contract
  * @notice CTokens which wrap an EIP-20 underlying and delegate to an implementation
  * @author Compound
  */
contract CErc20Delegator is CTokenInterface, CErc20Interface, CDelegatorInterface {
        /**
          * @notice Construct a new money market
          * @param underlying_ The address of the underlying asset
          * @param comptroller_ The address of the Comptroller
          * @param interestRateModel_ The address of the interest rate model
          * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
          * @param name_ ERC-20 name of this token
          * @param symbol_ ERC-20 symbol of this token
          * @param decimals_ ERC-20 decimal precision of this token
          * @param admin_ Address of the administrator of this token
          * @param implementation_ The address of the implementation the contract delegates to
          * @param becomeImplementationData The encoded args for becomeImplementation
          */
        constructor(address underlying_,
                ComptrollerInterface comptroller_,
                InterestRateModel interestRateModel_,
                uint initialExchangeRateMantissa_,
                string memory name_,
                string memory symbol_,
                uint8 decimals_,
                address payable admin_,
                address implementation_,
                bytes memory becomeImplementationData) public {
                // Creator of the contract is admin during initialization
                admin = msg.sender;

                // First delegate gets to initialize the delegator (i.e. storage contract)
                delegateTo(implementation_,

abi.encodeWithSignature("initialize(address,address,address,uint256,string,string,uint8)",
                        underlying_,
                        comptroller_,
                        interestRateModel_,
                        initialExchangeRateMantissa_,
                        name_,
                        symbol_,
                        decimals_));

                // New implementations always get set via the settor (post-initialize)
                _setImplementation(implementation_, false, becomeImplementationData);

                // Set the proper admin now that initialization is done
                admin = admin_;
                feeManager = admin_;
        }

        /**
          * @notice Called by the admin to update the implementation of the delegator
          * @param implementation_ The address of the new implementation for delegation
          * @param allowResign Flag to indicate whether to call _resignImplementation on the old
implementation
          * @param becomeImplementationData The encoded bytes data to be passed to
_becomeImplementation
          */
        function _setImplementation(
                address implementation_,
                bool allowResign,
                bytes memory becomeImplementationData
        ) public {
                require(msg.sender == admin, "CErc20Delegator::_setImplementation: Caller must be
admin");

                if (allowResign) {
                        delegateToImplementation(abi.encodeWithSignature("_resignImplementation()"));
                }

                address oldImplementation = implementation;
                implementation = implementation_;
```

```
        delegateToImplementation(abi.encodeWithSignature("_becomeImplementation(bytes)",
becomeImplementationData));

        emit NewImplementation(oldImplementation, implementation);
    }

    /**
     * @notice Sender supplies assets into the market and receives cTokens in exchange
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param mintAmount The amount of the underlying asset to supply
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function mint(uint mintAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("mint(uint256)",
mintAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Sender redeems cTokens in exchange for the underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemTokens The number of cTokens to redeem into underlying
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeem(uint redeemTokens) external returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("redeem(uint256)", redeemTokens));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemAmount The amount of underlying to redeem
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeemUnderlying(uint redeemAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("redeemUnderlying(uint256)", redeemAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Sender borrows assets from the protocol to their own address
     * @param borrowAmount The amount of the underlying asset to borrow
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function borrow(uint borrowAmount) external returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("borrow(uint256)", borrowAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Sender repays their own borrow
     * @param repayAmount The amount to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrow(uint repayAmount) external returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("repayBorrow(uint256)", repayAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @param borrower the account with the debt being payed off
     * @param repayAmount The amount to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("repayBorrowBehalf(address,uint256)", borrower,
repayAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice The sender liquidates the borrowers collateral.
     *  The collateral seized is transferred to the liquidator.
     * @param borrower The borrower of this cToken to be liquidated
     * @param cTokenCollateral The market in which to seize collateral from the borrower
     * @param repayAmount The amount of the underlying borrowed asset to repay
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function liquidateBorrow(
        address borrower,
```

```
            uint repayAmount,
            CTokenInterface cTokenCollateral
        ) external returns (uint) {
            bytes memory data = delegateToImplementation(
                abi.encodeWithSignature(
                    "liquidateBorrow(address,uint256,address)",
                    borrower,
                    repayAmount,
                    cTokenCollateral
                )
            );
            return abi.decode(data, (uint));
        }

        /**
          * @notice Transfer `amount` tokens from `msg.sender` to `dst`
          * @param dst The address of the destination account
          * @param amount The number of tokens to transfer
          * @return Whether or not the transfer succeeded
          */
        function transfer(address dst, uint amount) external returns (bool) {
            bytes memory data = delegateToImplementation(abi.encodeWithSignature("transfer(address,uint256)", dst, amount));
            return abi.decode(data, (bool));
        }

        /**
          * @notice Transfer `amount` tokens from `src` to `dst`
          * @param src The address of the source account
          * @param dst The address of the destination account
          * @param amount The number of tokens to transfer
          * @return Whether or not the transfer succeeded
          */
        function transferFrom(address src, address dst, uint256 amount) external returns (bool) {
            bytes memory data = delegateToImplementation(
                abi.encodeWithSignature("transferFrom(address,address,uint256)", src, dst, amount)
            );
            return abi.decode(data, (bool));
        }

        /**
          * @notice Approve `spender` to transfer up to `amount` from `src`
          * @dev This will overwrite the approval amount for `spender`
          *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
          * @param spender The address of the account which may transfer tokens
          * @param amount The number of tokens that are approved (-1 means infinite)
          * @return Whether or not the approval succeeded
          */
        function approve(address spender, uint256 amount) external returns (bool) {
            bytes memory data = delegateToImplementation(
                abi.encodeWithSignature("approve(address,uint256)", spender, amount)
            );
            return abi.decode(data, (bool));
        }

        /**
          * @notice Get the current allowance from `owner` for `spender`
          * @param owner The address of the account which owns the tokens to be spent
          * @param spender The address of the account which may transfer tokens
          * @return The number of tokens allowed to be spent (-1 means infinite)
          */
        function allowance(address owner, address spender) external view returns (uint) {
            bytes memory data = delegateToViewImplementation(
                abi.encodeWithSignature("allowance(address,address)", owner, spender)
            );
            return abi.decode(data, (uint));
        }

        /**
          * @notice Get the token balance of the `owner`
          * @param owner The address of the account to query
          * @return The number of tokens owned by `owner`
          */
        function balanceOf(address owner) external view returns (uint) {
            bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("balanceOf(address)", owner));
            return abi.decode(data, (uint));
        }

        /**
          * @notice Get the underlying balance of the `owner`
          * @dev This also accrues interest in a transaction
          * @param owner The address of the account to query
          * @return The amount of underlying owned by `owner`
          */
        function balanceOfUnderlying(address owner) external returns (uint) {
            bytes memory data =
```

```
delegateToImplementation(abi.encodeWithSignature("balanceOfUnderlying(address)", owner));
        return abi.decode(data, (uint));
    }

    function balanceOfUnderlyingView(address owner) external view returns (uint) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("balanceOfUnderlyingView(address)", owner)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @notice Get a snapshot of the account's balances, and the cached exchange rate
     * @dev This is used by comptroller to more efficiently perform liquidity checks.
     * @param account Address of the account to snapshot
     * @return (possible error, token balance, borrow balance, exchange rate mantissa)
     */
    function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("getAccountSnapshot(address)", account)
        );
        return abi.decode(data, (uint, uint, uint, uint));
    }

    /**
     * @notice Returns the current per-block borrow interest rate for this cToken
     * @return The borrow interest rate per block, scaled by 1e18
     */
    function borrowRatePerBlock() external view returns (uint) {
        bytes memory data =
delegateToViewImplementation(abi.encodeWithSignature("borrowRatePerBlock()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Returns the current per-block supply interest rate for this cToken
     * @return The supply interest rate per block, scaled by 1e18
     */
    function supplyRatePerBlock() external view returns (uint) {
        bytes memory data =
delegateToViewImplementation(abi.encodeWithSignature("supplyRatePerBlock()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Returns the current total borrows plus accrued interest
     * @return The total borrows with interest
     */
    function totalBorrowsCurrent() external returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("totalBorrowsCurrent()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrue interest to updated borrowIndex
     *           and then calculate account's borrow balance using the updated borrowIndex
     * @param account The address whose balance should be calculated after updating borrowIndex
     * @return The calculated balance
     */
    function borrowBalanceCurrent(address account) external returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("borrowBalanceCurrent(address)", account));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Return the borrow balance of account based on stored data
     * @param account The address whose balance should be calculated
     * @return The calculated balance
     */
    function borrowBalanceStored(address account) public view returns (uint) {
        bytes memory data = delegateToViewImplementation(
            abi.encodeWithSignature("borrowBalanceStored(address)", account)
        );
        return abi.decode(data, (uint));
    }

    /**
      * @notice Accrue interest then return the up-to-date exchange rate
      * @return Calculated exchange rate scaled by 1e18
      */
    function exchangeRateCurrent() public returns (uint) {
        bytes memory data =
delegateToImplementation(abi.encodeWithSignature("exchangeRateCurrent()"));
        return abi.decode(data, (uint));
```

```
    }

    /**
      * @notice Calculates the exchange rate from the underlying to the CToken
      * @dev This function does not accrue interest before calculating the exchange rate
      * @return Calculated exchange rate scaled by 1e18
      */
    function exchangeRateStored() public view returns (uint) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("exchangeRateStored()"));
        return abi.decode(data, (uint));
    }

    /**
      * @notice Get cash balance of this cToken in the underlying asset
      * @return The quantity of underlying asset owned by this contract
      */
    function getCash() external view returns (uint) {
        bytes memory data = delegateToViewImplementation(abi.encodeWithSignature("getCash()"));
        return abi.decode(data, (uint));
    }

    /**
      * @notice Applies accrued interest to total borrows and reserves.
      * @dev This calculates interest accrued from the last checkpointed block
      *   up to the current block and writes new checkpoint to storage.
      */
    function accrueInterest() public returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("accrueInterest()"));
        return abi.decode(data, (uint));
    }

    /**
      * @notice Transfers collateral tokens (this market) to the liquidator.
      * @dev Will fail unless called by another cToken during the process of liquidation.
      *  Its absolutely critical to use msg.sender as the borrowed cToken and not a parameter.
      * @param liquidator The account receiving seized collateral
      * @param borrower The account having collateral seized
      * @param seizeTokens The number of cTokens to seize
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function seize(address liquidator, address borrower, uint seizeTokens) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("seize(address,address,uint256)", liquidator, borrower, seizeTokens)
        );
        return abi.decode(data, (uint));
    }

    /*** Admin Functions ***/

    /**
      * @notice Begins transfer of admin rights.
      *  The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
      * @dev Admin function to begin change of admin.
      *  The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
      * @param newPendingAdmin New pending admin.
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPendingAdmin(address payable newPendingAdmin) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setPendingAdmin(address)", newPendingAdmin)
        );
        return abi.decode(data, (uint));
    }

    /**
      * @notice Sets a new comptroller for the market
      * @dev Admin function to set a new comptroller
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setComptroller(ComptrollerInterface newComptroller) public returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setComptroller(address)",
            newComptroller));
        return abi.decode(data, (uint));
    }

    function _setFeeManager(address payable _feeManager) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setFeeManager(address)",
            _feeManager));
        return abi.decode(data, (uint));
    }
```

```
    function flashLoan(
        address _receiver,
        address _reserve,
        uint256 _amount,
        bytes memory _params
    ) public returns (uint){
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("flashLoan(address,address,uint256,bytes)",
            _receiver, _reserve, _amount, _params));
        return abi.decode(data, (uint));
    }

    /**
     * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
     * @dev Admin function to accrue interest and set a new reserve factor
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _setReserveFactor(uint newReserveFactorMantissa) external returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setReserveFactor(uint256)", newReserveFactorMantissa)
        );
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
     * @dev Admin function for pending admin to accept role and update admin
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _acceptAdmin() external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_acceptAdmin()"));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrues interest and adds reserves by transferring from admin
     * @param addAmount Amount of reserves to add
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _addReserves(uint addAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_addReserves(uint256)", addAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrues interest and reduces reserves by transferring to admin
     * @param reduceAmount Amount of reduction to reserves
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _reduceReserves(uint reduceAmount) external returns (uint) {
        bytes memory data = delegateToImplementation(abi.encodeWithSignature("_reduceReserves(uint256)", reduceAmount));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Accrues interest and updates the interest rate model using _setInterestRateModelFresh
     * @dev Admin function to accrue interest and update the interest rate model
     * @param newInterestRateModel the new interest rate model to use
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
        bytes memory data = delegateToImplementation(
            abi.encodeWithSignature("_setInterestRateModel(address)",
            newInterestRateModel));
        return abi.decode(data, (uint));
    }

    /**
     * @notice Internal method to delegate execution to another contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     * @param callee The contract to delegatecall
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateTo(address callee, bytes memory data) internal returns (bytes memory) {
        (bool success, bytes memory returnData) = callee.delegatecall(data);
        assembly {
            if eq(success, 0) {
                revert(add(returnData, 0x20), returndatasize)
            }
        }
        return returnData;
```

```
    }

    /**
     * @notice Delegates execution to the implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateToImplementation(bytes memory data) public returns (bytes memory) {
        return delegateTo(implementation, data);
    }

    /**
     * @notice Delegates execution to an implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     *  There are an additional 2 prefix uints from the wrapper returndata, which we ignore since we make an extra hop.
     * @param data The raw data to delegatecall
     * @return The returned bytes from the delegatecall
     */
    function delegateToViewImplementation(bytes memory data) public view returns (bytes memory) {
        (bool success, bytes memory returnData) = address(this).staticcall(
            abi.encodeWithSignature("delegateToImplementation(bytes)", data)
        );
        assembly {
            if eq(success, 0) {
                revert(add(returnData, 0x20), returndatasize)
            }
        }
        return abi.decode(returnData, (bytes));
    }

    /**
     * @notice Delegates execution to an implementation contract
     * @dev It returns to the external caller whatever the implementation returns or forwards reverts
     */
    function() external payable {
        require(msg.value == 0, "CErc20Delegator:fallback: cannot send value to fallback");

        // delegate all other functions to current implementation
        (bool success,) = implementation.delegatecall(msg.data);

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 {revert(free_mem_ptr, returndatasize)}
            default {return (free_mem_ptr, returndatasize)}
        }
    }
}
```

CErc20Immutable.sol

```
pragma solidity ^0.5.9;

import "./CErc20.sol";

/**
 * @title Compound's CErc20Immutable Contract
 * @notice CTokens which wrap an EIP-20 underlying and are immutable
 * @author Compound
 */
contract CErc20Immutable is CErc20 {
    /**
     * @notice Construct a new money market
     * @param underlying_ The address of the underlying asset
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     * @param admin_ Address of the administrator of this token
     */
    constructor(address underlying_,
                ComptrollerInterface comptroller_,
                InterestRateModel interestRateModel_,
                uint initialExchangeRateMantissa_,
                string memory name_,
                string memory symbol_,
                uint8 decimals_,
                address payable admin_) public {
        // Creator of the contract is admin during initialization
        admin = msg.sender;
```

```
        // Initialize the market
        initialize(
            underlying_,
            comptroller_,
            interestRateModel_,
            initialExchangeRateMantissa_,
            name_,
            symbol_,
            decimals_
        );

        // Set the proper admin now that initialization is done
        admin = admin_;
    }
}
```

CNative.sol

```
pragma solidity ^0.5.9;

import "./CToken.sol";
import "./lib/NativeAddressLib.sol";

/**
  * @title Compound's CNative Contract
  * @notice CToken which wraps Ether
  * @author Compound
  */
contract CNative is CToken {
    /**
     * @notice Construct a new CNative money market
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     * @param admin_ Address of the administrator of this token
     */
    constructor(ComptrollerInterface comptroller_,
        InterestRateModel interestRateModel_,
        uint initialExchangeRateMantissa_,
        string memory name_,
        string memory symbol_,
        uint8 decimals_,
        address payable admin_) public {
        // Creator of the contract is admin during initialization
        admin = msg.sender;

        initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_, symbol_,
decimals_);

        // Set the proper admin now that initialization is done
        admin = admin_;
        underlying = NativeAddressLib.nativeAddress();

    }


    /*** User Interface ***/

    /**
     * @notice Sender supplies assets into the market and receives cTokens in exchange
     * @dev Reverts upon any failure
     */
    function mint() external payable {
        (uint err,) = mintInternal(msg.value);
        requireNoError(err, "mint failed");
    }

    /**
     * @notice Sender redeems cTokens in exchange for the underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemTokens The number of cTokens to redeem into underlying
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeem(uint redeemTokens) external returns (uint) {
        return redeemInternal(redeemTokens);
    }

    /**
     * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemAmount The amount of underlying to redeem
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeemUnderlying(uint redeemAmount) external returns (uint) {
```

```
        return redeemUnderlyingInternal(redeemAmount);
    }

    /**
      * @notice Sender borrows assets from the protocol to their own address
      * @param borrowAmount The amount of the underlying asset to borrow
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function borrow(uint borrowAmount) external returns (uint) {
        return borrowInternal(borrowAmount);
    }

    /**
      * @notice Sender repays their own borrow
      * @dev Reverts upon any failure
      */
    function repayBorrow() external payable {
        (uint err,) = repayBorrowInternal(msg.value);
        requireNoError(err, "repayBorrow failed");
    }

    /**
      * @notice Sender repays a borrow belonging to borrower
      * @dev Reverts upon any failure
      * @param borrower the account with the debt being payed off
      */
    function repayBorrowBehalf(address borrower) external payable {
        (uint err,) = repayBorrowBehalfInternal(borrower, msg.value);
        requireNoError(err, "repayBorrowBehalf failed");
    }

    /**
      * @notice The sender liquidates the borrowers collateral.
      *  The collateral seized is transferred to the liquidator.
      * @dev Reverts upon any failure
      * @param borrower The borrower of this cToken to be liquidated
      * @param cTokenCollateral The market in which to seize collateral from the borrower
      */
    function liquidateBorrow(address borrower, CToken cTokenCollateral) external payable {
        (uint err,) = liquidateBorrowInternal(borrower, msg.value, cTokenCollateral);
        requireNoError(err, "liquidateBorrow failed");
    }

    /**
      * @notice Send Ether to CNative to mint
      */
    function() external payable {
        (uint err,) = mintInternal(msg.value);
        requireNoError(err, "mint failed");
    }

    /*** Safe Token ***/

    /**
      * @notice Gets balance of this contract in terms of Ether, before this message
      * @dev This excludes the value of the current message, if any
      * @return The quantity of Ether owned by this contract
      */
    function getCashPrior() internal view returns (uint) {
        (MathError err, uint startingBalance) = subUInt(address(this).balance, msg.value);
        require(err == MathError.NO_ERROR);
        return startingBalance;
    }

    /**
      * @notice Perform the actual transfer in, which is a no-op
      * @param from Address sending the Ether
      * @param amount Amount of Ether being sent
      * @return The actual amount of Ether transferred
      */
    function doTransferIn(address from, uint amount) internal returns (uint) {
        // Sanity checks
        require(msg.sender == from, "sender mismatch");
        require(msg.value == amount, "value mismatch");
        return amount;
    }

    function doTransferOut(address payable to, uint amount) internal {
        /* Send the Ether, with minimal gas and revert on failure */
        to.transfer(amount);
    }

    function requireNoError(uint errCode, string memory message) internal pure {
        if (errCode == uint(Error.NO_ERROR)) {
            return;
        }
```

```
            bytes memory fullMessage = new bytes(bytes(message).length + 5);
            uint i;

            for (i = 0; i < bytes(message).length; i++) {
                fullMessage[i] = bytes(message)[i];
            }

            fullMessage[i + 0] = byte(uint8(32));
            fullMessage[i + 1] = byte(uint8(40));
            fullMessage[i + 2] = byte(uint8(48 + (errCode / 10)));
            fullMessage[i + 3] = byte(uint8(48 + (errCode % 10)));
            fullMessage[i + 4] = byte(uint8(41));

            require(errCode == uint(Error.NO_ERROR), string(fullMessage));
        }
    }
```

Comptroller.sol

```
pragma solidity ^0.5.9;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./lib/Exponential.sol";
import "./interface/PriceOracle.sol";
import "./interface/ComptrollerInterface.sol";
import "./interface/ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./token/RewardToken.sol";


/**
  * @title Compound's Comptroller Contract
  * @author Compound
  */
contract Comptroller is ComptrollerV4Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    /// @notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

    /// @notice Emitted when an account enters a market
    event MarketEntered(CToken cToken, address account);

    /// @notice Emitted when an account exits a market
    event MarketExited(CToken cToken, address account);

    /// @notice Emitted when close factor is changed by admin
    event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);

    /// @notice Emitted when a collateral factor is changed by admin
    event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFactorMantissa);

    /// @notice Emitted when liquidation incentive is changed by admin
    event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveMantissa);

    /// @notice Emitted when maxAssets is changed by admin
    event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);

    /// @notice Emitted when price oracle is changed
    event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);

    /// @notice Emitted when pause guardian is changed
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

    /// @notice Emitted when an action is paused globally
    event ActionPaused(string action, bool pauseState);

    /// @notice Emitted when an action is paused on a market
    event ActionPaused(CToken cToken, string action, bool pauseState);

    /// @notice Emitted when market comped status is changed
    event MarketComped(CToken cToken, bool isComped);

    /// @notice Emitted when COMP rate is changed
    event NewCompRate(uint oldCompRate, uint newCompRate);
    /// @notice Emitted when a new COMP speed is calculated for a market
    event CompSpeedUpdated(CToken indexed cToken, uint supplySpeed, uint borrowSpeed);

    /// @notice Emitted when COMP is distributed to a supplier
    event DistributedSupplierComp(
        CToken indexed cToken,
        address indexed supplier,
        uint compDelta,
        uint compSupplyIndex);

    /// @notice Emitted when COMP is distributed to a borrower
```

```
    event DistributedBorrowerComp(
        CToken indexed cToken,
        address indexed borrower,
        uint compDelta,
        uint compBorrowIndex);

    /// @notice Emitted when borrow cap for a cToken is changed
    event NewBorrowCap(CToken indexed cToken, uint newBorrowCap);

    /// @notice Emitted when borrow cap guardian is changed
    event NewBorrowCapGuardian(address oldBorrowCapGuardian, address
newBorrowCapGuardian);

    /// @notice The threshold above which the flywheel transfers COMP, in wei
    uint public constant compClaimThreshold = 0.001e18;

    /// @notice The initial COMP index for a market
    uint224 public constant compInitialIndex = 1e36;

    // closeFactorMantissa must be strictly greater than this value
    uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    // No collateralFactorMantissa may exceed this value
    uint internal constant collateralFactorMaxMantissa = 0.9e18; // 0.9

    // liquidationIncentiveMantissa must be no less than this value
    uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0

    // liquidationIncentiveMantissa must be no greater than this value
    uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

    uint public constant DEFAULT_TOKEN_WEIGHT = 100;
    uint public constant MAX_TOKEN_WEIGHT = 500;

    address private riskControl;

    constructor() public {
        admin = msg.sender;
        rewardAddress = 0xc00e94Cb662C3520282E6f5717214004A7f26888;
    }

    /*** Assets You Are In ***/

    /**
     * @notice Returns the assets an account has entered
     * @param account The address of the account to pull assets for
     * @return A dynamic list with the assets the account has entered
     */
    function getAssetsIn(address account) external view returns (CToken[] memory) {
        CToken[] memory assetsIn = accountAssets[account];

        return assetsIn;
    }

    /**
     * @notice Returns whether the given account is entered in the given asset
     * @param account The address of the account to check
     * @param cToken The cToken to check
     * @return True if the account is in the asset, otherwise false.
     */
    function checkMembership(address account, CToken cToken) external view returns (bool) {
        return markets[address(cToken)].accountMembership[account];
    }

    /**
     * @notice Add assets to be included in account liquidity calculation
     * @param cTokens The list of addresses of the cToken markets to be enabled
     * @return Success indicator for whether each corresponding market was entered
     */
    function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
        uint len = cTokens.length;

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }

        return results;
    }

    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
```

```
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */

    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }
        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }
        if (accountAssets[borrower].length >= maxAssets) {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }
        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //    this avoids having to iterate through the list for the most common use cases
        //    that is, only when we need to perform liquidity checks
        //    and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);
        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *  or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed,) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed");
        // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE,
FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION,
allowed);
        }

        Market storage marketToExit = markets[address(cToken)];

        /* Return true if the sender is not already ‘in’ the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender];
        storedList[assetIndex] = storedList[storedList.length - 1];
```

```
            storedList.length--;

            emit MarketExited(cToken, msg.sender);

            return uint(Error.NO_ERROR);
        }

        /*** Policy Hooks ***/

        /**
          * @notice Checks if the account should be allowed to mint tokens in the given market
          * @param cToken The market to verify the mint against
          * @param minter The account which would get the minted tokens
          * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
          * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
          */
        function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
            // Pausing is a very serious situation - we revert to sound the alarms
            require(!mintGuardianPaused[cToken], "mint is paused");

            // Shh - currently unused
            minter;
            mintAmount;

            if (!markets[cToken].isListed) {
                return uint(Error.MARKET_NOT_LISTED);
            }

            // Keep the flywheel moving
            updateCompSupplyIndex(cToken);
            distributeSupplierComp(cToken, minter, false);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Validates mint and reverts on rejection. May emit logs.
          * @param cToken Asset being minted
          * @param minter The address minting the tokens
          * @param actualMintAmount The amount of the underlying asset being minted
          * @param mintTokens The number of tokens being minted
          */
        function mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens) external {
            // Shh - currently unused
            cToken;
            minter;
            actualMintAmount;
            mintTokens;

            // Shh - we don't ever want this hook to be marked pure
            if (false) {
                maxAssets = maxAssets;
            }
        }

        /**
          * @notice Checks if the account should be allowed to redeem tokens in the given market
          * @param cToken The market to verify the redeem against
          * @param redeemer The account which would redeem the tokens
          * @param redeemTokens The number of cTokens to exchange for the underlying asset in the market
          * @return 0 if the redeem is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
          */
        function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uint) {
            uint allowed = redeemAllowedInternal(cToken, redeemer, redeemTokens);
            if (allowed != uint(Error.NO_ERROR)) {
                return allowed;
            }

            // Keep the flywheel moving
            updateCompSupplyIndex(cToken);
            distributeSupplierComp(cToken, redeemer, false);

            return uint(Error.NO_ERROR);
        }

        function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens) internal view returns (uint) {
            if (!markets[cToken].isListed) {
                return uint(Error.MARKET_NOT_LISTED);
            }

            /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
```

```
            if (!markets[cToken].accountMembership[redeemer]) {
                return uint(Error.NO_ERROR);
            }

            /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
            (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(
                redeemer,
                CToken(cToken),
                redeemTokens,
                0);
            if (err != Error.NO_ERROR) {
                return uint(err);
            }
            if (shortfall > 0) {
                return uint(Error.INSUFFICIENT_LIQUIDITY);
            }

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Validates redeem and reverts on rejection. May emit logs.
          * @param cToken Asset being redeemed
          * @param redeemer The address redeeming the tokens
          * @param redeemAmount The amount of the underlying asset being redeemed
          * @param redeemTokens The number of tokens being redeemed
          */
        function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens)
external {
            // Shh - currently unused
            cToken;
            redeemer;

            // Require tokens is zero or amount is also zero
            if (redeemTokens == 0 && redeemAmount > 0) {
                revert("redeemTokens zero");
            }
        }

        /**
          * @notice Checks if the account should be allowed to borrow the underlying asset of the given
market
          * @param cToken The market to verify the borrow against
          * @param borrower The account which would borrow the asset
          * @param borrowAmount The amount of underlying the account would borrow
          * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See
ErrorReporter.sol)
          */
        function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns
(uint) {
            // Pausing is a very serious situation - we revert to sound the alarms
            require(!borrowGuardianPaused[cToken], "borrow is paused");

            if (!markets[cToken].isListed) {
                return uint(Error.MARKET_NOT_LISTED);
            }

            if (!markets[cToken].accountMembership[borrower]) {
                // only cTokens may call borrowAllowed if borrower not in market
                require(msg.sender == cToken, "sender must be cToken");

                // attempt to add borrower to the market
                Error err = addToMarketInternal(CToken(msg.sender), borrower);
                if (err != Error.NO_ERROR) {
                    return uint(err);
                }

                // it should be impossible to break the important invariant
                assert(markets[cToken].accountMembership[borrower]);
            }

            if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
                return uint(Error.PRICE_ERROR);
            }


            uint borrowCap = borrowCaps[cToken];
            // Borrow cap of 0 corresponds to unlimited borrowing
            if (borrowCap != 0) {
                uint totalBorrows = CToken(cToken).totalBorrows();
                (MathError mathErr, uint nextTotalBorrows) = addUInt(totalBorrows, borrowAmount);
                require(mathErr == MathError.NO_ERROR, "total borrows overflow");
                require(nextTotalBorrows < borrowCap, "market borrow cap reached");
            }

            (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(
                borrower,
```

```
                CToken(cToken),
                0,
                borrowAmount);
        if (err != Error.NO_ERROR) {
                return uint(err);
        }
        if (shortfall > 0) {
                return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa : CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
}

/**
  * @notice Validates borrow and reverts on rejection. May emit logs.
  * @param cToken Asset whose underlying is being borrowed
  * @param borrower The address borrowing the underlying
  * @param borrowAmount The amount of the underlying asset requested to borrow
  */
function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
        // Shh - currently unused
        cToken;
        borrower;
        borrowAmount;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
                maxAssets = maxAssets;
        }
}

/**
  * @notice Checks if the account should be allowed to repay a borrow in the given market
  * @param cToken The market to verify the repay against
  * @param payer The account which would repay the asset
  * @param borrower The account which would borrowed the asset
  * @param repayAmount The amount of the underlying asset the account would repay
  * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
  */
function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        payer;
        borrower;
        repayAmount;

        if (!markets[cToken].isListed) {
                return uint(Error.MARKET_NOT_LISTED);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa : CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
}

/**
  * @notice Validates repayBorrow and reverts on rejection. May emit logs.
  * @param cToken Asset being repaid
  * @param payer The address repaying the borrow
  * @param borrower The address of the borrower
  * @param actualRepayAmount The amount of underlying being repaid
  */
function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint actualRepayAmount,
        uint borrowerIndex) external {
        // Shh - currently unused
        cToken;
        payer;
        borrower;
        actualRepayAmount;
        borrowerIndex;

        // Shh - we don't ever want this hook to be marked pure
```

```
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the liquidation should be allowed to occur
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        liquidator;

        if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* The borrower must have shortfall in order to be liquidatable */
        (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall == 0) {
            return uint(Error.INSUFFICIENT_SHORTFALL);
        }

        /* The liquidator may not repay more than what is allowed by the closeFactor */
        uint borrowBalance = CToken(cTokenBorrowed).borrowBalanceStored(borrower);
        (MathError mathErr, uint maxClose) = mulScalarTruncate(Exp({mantissa :
closeFactorMantissa}), borrowBalance);
        if (mathErr != MathError.NO_ERROR) {
            return uint(Error.MATH_ERROR);
        }
        if (repayAmount > maxClose) {
            return uint(Error.TOO_MUCH_REPAY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint actualRepayAmount,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenBorrowed;
        cTokenCollateral;
        liquidator;
        borrower;
        actualRepayAmount;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
```

```
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!seizeGuardianPaused, "seize is paused");

        // Shh - currently unused
        seizeTokens;

        if (!markets[cTokenCollateral].isListed || !markets[cTokenBorrowed].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
            return uint(Error.COMPTROLLER_MISMATCH);
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cTokenCollateral);
        distributeSupplierComp(cTokenCollateral, borrower, false);
        distributeSupplierComp(cTokenCollateral, liquidator, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenCollateral;
        cTokenBorrowed;
        liquidator;
        borrower;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // Currently the only consideration is whether or not
        //  the src is allowed to redeem this many tokens
        uint allowed = redeemAllowedInternal(cToken, src, transferTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, src, false);
        distributeSupplierComp(cToken, dst, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates transfer and reverts on rejection. May emit logs.
     * @param cToken Asset being transferred
     * @param src The account which sources the tokens
```

```
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     */
    function transferVerify(address cToken, address src, address dst, uint transferTokens) external {
        // Shh - currently unused
        cToken;
        src;
        dst;
        transferTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /*** Liquidity/Liquidation Calculations ***/

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
     *  Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     *  whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code (semi-opaque),
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account,
CToken(0), 0, 0);

        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code,
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidityInternal(address account) internal view returns (Error, uint, uint) {
        return getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were
redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(
            account,
            CToken(cTokenModify),
            redeemTokens,
            borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were
redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
```

```
        * @param account The account to determine liquidity for
        * @param redeemTokens The number of tokens to hypothetically redeem
        * @param borrowAmount The amount of underlying to hypothetically borrow
        * @dev Note that we calculate the exchangeRateStored for each collateral cToken using stored data,
        *  without calculating accumulated interest.
        * @return (possible error code,
                    hypothetical account liquidity in excess of collateral requirements,
        *           hypothetical account shortfall below collateral requirements)
        */
    function getHypotheticalAccountLiquidityInternal(
            address account,
            CToken cTokenModify,
            uint redeemTokens,
            uint borrowAmount) internal view returns (Error, uint, uint) {

            AccountLiquidityLocalVars memory vars;
            // Holds all our calculation results
            uint oErr;
            MathError mErr;

            // For each asset the account is in
            CToken[] memory assets = accountAssets[account];
            for (uint i = 0; i < assets.length; i++) {
                CToken asset = assets[i];

                // Read the balances and exchange rate from the cToken
                (oErr,
                vars.cTokenBalance,
                vars.borrowBalance,
                vars.exchangeRateMantissa) = asset.getAccountSnapshot(account);
                if (oErr != 0) {// semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
                    return (Error.SNAPSHOT_ERROR, 0, 0);
                }
                vars.collateralFactor = Exp({mantissa : markets[address(asset)].collateralFactorMantissa});
                vars.exchangeRate = Exp({mantissa : vars.exchangeRateMantissa});

                // Get the normalized price of the asset
                vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
                if (vars.oraclePriceMantissa == 0) {
                    return (Error.PRICE_ERROR, 0, 0);
                }
                vars.oraclePrice = Exp({mantissa : vars.oraclePriceMantissa});

                // Pre-compute a conversion factor from tokens -> ether (normalized price value)
                (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate, vars.oraclePrice);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // sumCollateral += tokensToDenom * cTokenBalance
                (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(
                    vars.tokensToDenom,
                    vars.cTokenBalance,
                    vars.sumCollateral);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // sumBorrowPlusEffects += oraclePrice * borrowBalance
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(
                    vars.oraclePrice,
                    vars.borrowBalance,
                    vars.sumBorrowPlusEffects);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // Calculate effects of interacting with cTokenModify
                if (asset == cTokenModify) {
                    // redeem effect
                    // sumBorrowPlusEffects += tokensToDenom * redeemTokens
                    (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(
                        vars.tokensToDenom,
                        redeemTokens,
                        vars.sumBorrowPlusEffects);
                    if (mErr != MathError.NO_ERROR) {
                        return (Error.MATH_ERROR, 0, 0);
                    }

                    // borrow effect
                    // sumBorrowPlusEffects += oraclePrice * borrowAmount
                    (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(
                        vars.oraclePrice,
```

```
                        borrowAmount,
                        vars.sumBorrowPlusEffects);
                    if (mErr != MathError.NO_ERROR) {
                        return (Error.MATH_ERROR, 0, 0);
                    }
                }
            }

            // These are safe, as the underflow condition is checked first
            if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
                return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
            } else {
                return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
            }
        }

        /**
          * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
          * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
          * @param cTokenBorrowed The address of the borrowed cToken
          * @param cTokenCollateral The address of the collateral cToken
          * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral tokens
          * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
          */
        function liquidateCalculateSeizeTokens(
            address cTokenBorrowed,
            address cTokenCollateral,
            uint actualRepayAmount
        ) external view returns (uint, uint) {
            /* Read oracle prices for borrowed and collateral markets */
            uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
            uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
            if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
                return (uint(Error.PRICE_ERROR), 0);
            }


            /*
              * Get the exchange rate and calculate the number of collateral tokens to seize:
              *  seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
              *  seizeTokens = seizeAmount / exchangeRate
              *  = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
              */
            uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored();


            // Note: reverts on error
            uint seizeTokens;
            Exp memory numerator;
            Exp memory denominator;
            Exp memory ratio;
            MathError mathErr;

            (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
            if (mathErr != MathError.NO_ERROR) {
                return (uint(Error.MATH_ERROR), 0);
            }

            (mathErr, denominator) = mulExp(priceCollateralMantissa, exchangeRateMantissa);
            if (mathErr != MathError.NO_ERROR) {
                return (uint(Error.MATH_ERROR), 0);
            }

            (mathErr, ratio) = divExp(numerator, denominator);
            if (mathErr != MathError.NO_ERROR) {
                return (uint(Error.MATH_ERROR), 0);
            }

            (mathErr, seizeTokens) = mulScalarTruncate(ratio, actualRepayAmount);
            if (mathErr != MathError.NO_ERROR) {
                return (uint(Error.MATH_ERROR), 0);
            }

            return (uint(Error.NO_ERROR), seizeTokens);
        }

        /*** Admin Functions ***/

        /**
          * @notice Sets a new price oracle for the comptroller
          * @dev Admin function to set a new price oracle
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
```

```
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);

        return uint(Error.NO_ERROR);
    }

    function getRiskControl() external returns (address){
        return riskControl;
    }

    function _setRiskControl(address _address) external {
        require(msg.sender == admin, "!admin");
        riskControl = _address;
    }

    /**
      * @notice Sets the closeFactor used when liquidating borrows
      * @dev Admin function to set closeFactor
      * @param newCloseFactorMantissa New close factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
        }

        Exp memory newCloseFactorExp = Exp({mantissa : newCloseFactorMantissa});
        Exp memory lowLimit = Exp({mantissa : closeFactorMinMantissa});
        if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        Exp memory highLimit = Exp({mantissa : closeFactorMaxMantissa});
        if (lessThanExp(highLimit, newCloseFactorExp)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets the collateralFactor for a market
      * @dev Admin function to set per-market collateralFactor
      * @param cToken The market to set the factor on
      * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

        // Verify market is listed
        Market storage market = markets[address(cToken)];
        if (!market.isListed) {
            return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
        }

        Exp memory newCollateralFactorExp = Exp({mantissa : newCollateralFactorMantissa});

        // Check collateral factor <= 0.9
        Exp memory highLimit = Exp({mantissa : collateralFactorMaxMantissa});
        if (lessThanExp(highLimit, newCollateralFactorExp)) {
```

```
                return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
            }

            // If collateral factor != 0, fail if price == 0
            if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
                return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
            }

            // Set market's collateral factor to new collateral factor, remember old value
            uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
            market.collateralFactorMantissa = newCollateralFactorMantissa;

            // Emit event with asset, old collateral factor, and new collateral factor
            emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Sets maxAssets which controls how many markets can be entered
          * @dev Admin function to set maxAssets
          * @param newMaxAssets New max assets
          * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
          */
        function _setMaxAssets(uint newMaxAssets) external returns (uint) {
            // Check caller is admin
            if (msg.sender != admin) {
                return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
            }

            uint oldMaxAssets = maxAssets;
            maxAssets = newMaxAssets;
            emit NewMaxAssets(oldMaxAssets, newMaxAssets);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Sets liquidationIncentive
          * @dev Admin function to set liquidationIncentive
          * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
          * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
          */
        function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint) {
            // Check caller is admin
            if (msg.sender != admin) {
                return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
            }

            // Check de-scaled min <= newLiquidationIncentive <= max
            Exp memory newLiquidationIncentive = Exp({mantissa : newLiquidationIncentiveMantissa});
            Exp memory minLiquidationIncentive = Exp({mantissa : liquidationIncentiveMinMantissa});
            if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
                return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
            }

            Exp memory maxLiquidationIncentive = Exp({mantissa : liquidationIncentiveMaxMantissa});
            if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
                return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
            }

            // Save current value for use in log
            uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;

            // Set liquidation incentive to new incentive
            liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;

            // Emit event with old incentive, new incentive
            emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Add the market to the markets mapping and set it as listed
          * @dev Admin function to set isListed and add support for the market
          * @param cToken The address of the market (token) to list
```

```
        * @return uint 0=success, otherwise a failure. (See enum Error for details)
        */
    function _supportMarket(CToken cToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(cToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        cToken.isCToken();
        // Sanity check to make sure its really a CToken

        markets[address(cToken)] = Market({
        isListed : true,
        isComped : false,
        collateralFactorMantissa : 0,
        weight : DEFAULT_TOKEN_WEIGHT
        });

        _addMarketInternal(address(cToken));

        emit MarketListed(cToken);

        return uint(Error.NO_ERROR);
    }

    function _addMarketInternal(address cToken) internal {
        for (uint i = 0; i < allMarkets.length; i ++) {
            require(allMarkets[i] != CToken(cToken), "market already added");
        }
        allMarkets.push(CToken(cToken));
    }


    /**
      * @notice Set the given borrow caps for the given cToken markets.
      *         Borrowing that brings total borrows to or above borrow cap will revert.
      * @dev Admin or borrowCapGuardian function to set the borrow caps.
      *   A borrow cap of 0 corresponds to unlimited borrowing.
      * @param cTokens The addresses of the markets (tokens) to change the borrow caps for
      * @param newBorrowCaps The new borrow cap values in underlying to be set.
      *   A value of 0 corresponds to unlimited borrowing.
      */
    function _setMarketBorrowCaps(CToken[] calldata cTokens, uint[] calldata newBorrowCaps) external {
        require(msg.sender == admin || msg.sender == borrowCapGuardian,
            "only admin or borrow cap guardian can set borrow caps");

        uint numMarkets = cTokens.length;
        uint numBorrowCaps = newBorrowCaps.length;

        require(numMarkets != 0 && numMarkets == numBorrowCaps, "invalid input");

        for (uint i = 0; i < numMarkets; i++) {
            borrowCaps[address(cTokens[i])] = newBorrowCaps[i];
            emit NewBorrowCap(cTokens[i], newBorrowCaps[i]);
        }
    }

    /**
      * @notice Admin function to change the Borrow Cap Guardian
      * @param newBorrowCapGuardian The address of the new Borrow Cap Guardian
      */
    function _setBorrowCapGuardian(address newBorrowCapGuardian) external {
        require(msg.sender == admin, "only admin can set borrow cap guardian");

        // Save current value for inclusion in log
        address oldBorrowCapGuardian = borrowCapGuardian;

        // Store borrowCapGuardian with value newBorrowCapGuardian
        borrowCapGuardian = newBorrowCapGuardian;

        // Emit NewBorrowCapGuardian(OldBorrowCapGuardian, NewBorrowCapGuardian)
        emit NewBorrowCapGuardian(oldBorrowCapGuardian, newBorrowCapGuardian);
    }

    /**
      * @notice Admin function to change the Pause Guardian
      * @param newPauseGuardian The address of the new Pause Guardian
      * @return uint 0=success, otherwise a failure. (See enum Error for details)
      */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
```

```
                return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
            }

            // Save current value for inclusion in log
            address oldPauseGuardian = pauseGuardian;

            // Store pauseGuardian with value newPauseGuardian
            pauseGuardian = newPauseGuardian;

            // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
            emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

            return uint(Error.NO_ERROR);
        }

        function _setMintPaused(CToken cToken, bool state) public returns (bool) {
            require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
            require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
            require(msg.sender == admin || state == true, "only admin can unpause");

            mintGuardianPaused[address(cToken)] = state;
            emit ActionPaused(cToken, "Mint", state);
            return state;
        }

        function _setBorrowPaused(CToken cToken, bool state) public returns (bool) {
            require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
            require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
            require(msg.sender == admin || state == true, "only admin can unpause");

            borrowGuardianPaused[address(cToken)] = state;
            emit ActionPaused(cToken, "Borrow", state);
            return state;
        }

        function _setTransferPaused(bool state) public returns (bool) {
            require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
            require(msg.sender == admin || state == true, "only admin can unpause");

            transferGuardianPaused = state;
            emit ActionPaused("Transfer", state);
            return state;
        }

        function _setSeizePaused(bool state) public returns (bool) {
            require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
            require(msg.sender == admin || state == true, "only admin can unpause");

            seizeGuardianPaused = state;
            emit ActionPaused("Seize", state);
            return state;
        }

        function _become(Unitroller unitroller) public {
            require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
            require(unitroller._acceptImplementation() == 0, "change not authorized");
        }

        /**
          * @notice Checks caller is admin, or this contract is becoming the new implementation
          */
        function adminOrInitializing() internal view returns (bool) {
            return msg.sender == admin || msg.sender == comptrollerImplementation;
        }

        /*** Comp Distribution ***/

        /**
          * @notice Recalculate and update COMP speeds for all COMP markets
          */
        function refreshCompSpeeds() public {
            require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");
            refreshCompSpeedsInternal();
        }

        function refreshCompSpeedsInternal() internal {
            CToken[] memory allMarkets_ = allMarkets;

            for (uint i = 0; i < allMarkets_.length; i++) {
                CToken cToken = allMarkets_[i];
                Exp memory borrowIndex = Exp({mantissa : cToken.borrowIndex()});
                updateCompSupplyIndex(address(cToken));
```

```
                updateCompBorrowIndex(address(cToken), borrowIndex);
            }

            Exp memory totalUtility = Exp({mantissa : 0});
            Exp[] memory utilities = new Exp[](allMarkets_.length);
            for (uint i = 0; i < allMarkets_.length; i++) {
                CToken cToken = allMarkets_[i];
                Market memory market = markets[address(cToken)];
                if (market.isComped) {
                    Exp memory assetPrice = Exp({mantissa : oracle.getUnderlyingPrice(cToken)});
                    Exp memory utility = mul_(assetPrice, cToken.totalBorrows());
                    utilities[i] = div_(mul_(utility, market.weight), DEFAULT_TOKEN_WEIGHT);
                    totalUtility = add_(totalUtility, utility);
                }
            }

            for (uint i = 0; i < allMarkets_.length; i++) {
                CToken cToken = allMarkets[i];
                uint comSupplyRate = div_(mul_(compRate, comSupplyRatio), expScale);
                uint comBorrowRate = div_(mul_(compRate, comBorrowRatio), expScale);
                uint newSupplySpeed = totalUtility.mantissa > 0 ? mul_(comSupplyRate,
div_(utilities[i], totalUtility)) : 0;
                uint newBorrowSpeed = totalUtility.mantissa > 0 ? mul_(comBorrowRate,
div_(utilities[i], totalUtility)) : 0;

                compSpeeds[address(cToken)] = newSupplySpeed;
                compBorrowSpeeds[address(cToken)] = newBorrowSpeed;

                emit CompSpeedUpdated(cToken, newSupplySpeed, newBorrowSpeed);
            }
        }

        /**
         * @notice Accrue COMP to the market by updating the supply index
         * @param cToken The market whose supply index to update
         */
        function updateCompSupplyIndex(address cToken) internal {
            CompMarketState storage supplyState = compSupplyState[cToken];
            uint supplySpeed = compSpeeds[cToken];
            uint blockNumber = getBlockNumber();
            uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
            if (deltaBlocks > 0 && supplySpeed > 0) {
                uint supplyTokens = CToken(cToken).totalSupply();
                uint compAccrued = mul_(deltaBlocks, supplySpeed);
                Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens) :
Double({mantissa : 0});
                Double memory index = add_(Double({mantissa : supplyState.index}), ratio);
                compSupplyState[cToken] = CompMarketState({
                index : safe224(index.mantissa, "new index exceeds 224 bits"),
                block : safe32(blockNumber, "block number exceeds 32 bits")
                });
            } else if (deltaBlocks > 0) {
                supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
            }
        }

        /**
         * @notice Accrue COMP to the market by updating the borrow index
         * @param cToken The market whose borrow index to update
         */
        function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
            CompMarketState storage borrowState = compBorrowState[cToken];
            uint borrowSpeed = compBorrowSpeeds[cToken];
            uint blockNumber = getBlockNumber();
            uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
            if (deltaBlocks > 0 && borrowSpeed > 0) {
                uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
                uint compAccrued = mul_(deltaBlocks, borrowSpeed);
                Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) :
Double({mantissa : 0});
                Double memory index = add_(Double({mantissa : borrowState.index}), ratio);
                compBorrowState[cToken] = CompMarketState({
                index : safe224(index.mantissa, "new index exceeds 224 bits"),
                block : safe32(blockNumber, "block number exceeds 32 bits")
                });
            } else if (deltaBlocks > 0) {
                borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
            }
        }

        /**
         * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
         * @param cToken The market in which the supplier is interacting
         * @param supplier The address of the supplier to distribute COMP to
         */
        function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
            CompMarketState storage supplyState = compSupplyState[cToken];
```

```
        Double memory supplyIndex = Double({mantissa : supplyState.index});
        Double memory supplierIndex = Double({mantissa : compSupplierIndex[cToken][supplier]});
        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
            supplierIndex.mantissa = compInitialIndex;
        }

        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
        uint supplierTokens = CToken(cToken).balanceOf(supplier);
        uint supplierDelta = mul_(supplierTokens, deltaIndex);
        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
        compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ? 0 : compClaimThreshold);
        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
    }

    /**
      * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
      * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
      * @param cToken The market in which the borrower is interacting
      * @param borrower The address of the borrower to distribute COMP to
      */
    function distributeBorrowerComp(
        address cToken,
        address borrower,
        Exp memory marketBorrowIndex,
        bool distributeAll
    ) internal {
        CompMarketState storage borrowState = compBorrowState[cToken];
        Double memory borrowIndex = Double({mantissa : borrowState.index});
        Double memory borrowerIndex = Double({mantissa : compBorrowerIndex[cToken][borrower]});
        compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

        if (borrowerIndex.mantissa > 0) {
            Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
            uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowIndex);
            uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
            uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
            compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : compClaimThreshold);
            emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
        }
    }

    /**
      * @notice Transfer COMP to the user, if they are above the threshold
      * @dev Note: If there is not enough COMP, we do not perform the transfer all.
      * @param user The address of the user to transfer COMP to
      * @param userAccrued The amount of COMP to (possibly) transfer
      * @return The amount of COMP which was NOT transferred to the user
      */
    function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
        if (userAccrued >= threshold && userAccrued > 0) {
            RewardToken comp = RewardToken(rewardAddress);
            uint compRemaining = comp.balanceOf(address(this));
            if (userAccrued <= compRemaining) {
                comp.transfer(user, userAccrued);
                return 0;
            }
        }
        return userAccrued;
    }

    /**
      * @notice Claim all the comp accrued by holder in all markets
      * @param holder The address to claim COMP for
      */
    function claimComp(address holder) public {
        return claimComp(holder, allMarkets);
    }

    /**
      * @notice Claim all the comp accrued by holder in the specified markets
      * @param holder The address to claim COMP for
      * @param cTokens The list of markets to claim COMP in
      */
    function claimComp(address holder, CToken[] memory cTokens) public {
        address[] memory holders = new address[](1);
        holders[0] = holder;
        claimComp(holders, cTokens, true, true);
    }
```

```
    /**
      * @notice Claim all comp accrued by the holders
      * @param holders The addresses to claim COMP for
      * @param cTokens The list of markets to claim COMP in
      * @param borrowers Whether or not to claim COMP earned by borrowing
      * @param suppliers Whether or not to claim COMP earned by supplying
      */
    function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public {
        for (uint i = 0; i < cTokens.length; i++) {
            CToken cToken = cTokens[i];
            require(markets[address(cToken)].isListed, "market must be listed");
            if (borrowers == true) {
                Exp memory borrowIndex = Exp({mantissa : cToken.borrowIndex()});
                updateCompBorrowIndex(address(cToken), borrowIndex);
                for (uint j = 0; j < holders.length; j++) {
                    distributeBorrowerComp(address(cToken), holders[j], borrowIndex, true);
                }
            }
            if (suppliers == true) {
                updateCompSupplyIndex(address(cToken));
                for (uint j = 0; j < holders.length; j++) {
                    distributeSupplierComp(address(cToken), holders[j], true);
                }
            }
        }
    }

    /*** Comp Distribution Admin ***/

    /**
      * @notice Set the amount of COMP distributed per block
      * @param compRate_ The amount of COMP wei per block to distribute
      */
    function _setCompRate(uint compRate_) public {
        require(adminOrInitializing() || msg.sender == reservoir, "only admin and reservoir can change comp rate");

        uint oldRate = compRate;
        compRate = compRate_;
        emit NewCompRate(oldRate, compRate_);

        refreshCompSpeedsInternal();
    }

    function _setReservoir(address _address) public {
        require(adminOrInitializing(), "only admin can change reservoir");
        reservoir = _address;
    }

    function _setRewardAddress(address _address) public {
        require(adminOrInitializing(), "only admin can change comp address");
        rewardAddress = _address;

    }

    function _setComBorrowRatio(uint compRate_) public {
        require(adminOrInitializing(), "only admin can change comp rate");
        comBorrowRatio = compRate_;
        refreshCompSpeedsInternal();
    }

    function _setComSupplyRatio(uint compRate_) public {
        require(adminOrInitializing(), "only admin can change comp rate");
        comSupplyRatio = compRate_;
        refreshCompSpeedsInternal();
    }

    /**
      * @notice Add markets to compMarkets, allowing them to earn COMP in the flywheel
      * @param cTokens The addresses of the markets to add
      */
    function _addCompMarkets(address[] memory cTokens) public {
        require(adminOrInitializing(), "only admin can add comp market");

        for (uint i = 0; i < cTokens.length; i++) {
            _addCompMarketInternal(cTokens[i]);
        }

        refreshCompSpeedsInternal();
    }

    function _setCompWeight(address cToken, uint _weight) public {
        require(adminOrInitializing(), "only admin can set weight");
        Market storage market = markets[cToken];
        require(market.isListed == true, "comp market is not listed");
```

```
            require(market.isComped == true, "comp market is not added");
            require(_weight < MAX_TOKEN_WEIGHT, "weight < max");
            market.weight = _weight;

        }

        function _addCompMarketInternal(address cToken) internal {
            Market storage market = markets[cToken];
            require(market.isListed == true, "comp market is not listed");
            require(market.isComped == false, "comp market already added");

            market.isComped = true;
            market.weight = DEFAULT_TOKEN_WEIGHT;
            emit MarketComped(CToken(cToken), true);

            if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
                compSupplyState[cToken] = CompMarketState({
                index : compInitialIndex,
                block : safe32(getBlockNumber(), "block number exceeds 32 bits")
                });
            }

            if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
                compBorrowState[cToken] = CompMarketState({
                index : compInitialIndex,
                block : safe32(getBlockNumber(), "block number exceeds 32 bits")
                });
            }
        }

        /**
          * @notice Remove a market from compMarkets, preventing it from earning COMP in the
flywheel
          * @param cToken The address of the market to drop
          */
        function _dropCompMarket(address cToken) public {
            require(msg.sender == admin, "only admin can drop comp market");

            Market storage market = markets[cToken];
            require(market.isComped == true, "market is not a comp market");

            market.isComped = false;
            emit MarketComped(CToken(cToken), false);

            refreshCompSpeedsInternal();
        }

        /**
          * @notice Return all of the markets
          * @dev The automatic getter may be used to access an individual market.
          * @return The list of market addresses
          */
        function getAllMarkets() public view returns (CToken[] memory) {
            return allMarkets;
        }

        function getBlockNumber() public view returns (uint) {
            return block.number;
        }

}
```

CToken.sol

```
pragma solidity ^0.5.9;

import "./interface/ComptrollerInterface.sol";
import "./interface/CTokenInterface.sol";
import "./ErrorReporter.sol";
import "./lib/Exponential.sol";
import "./lib/IERC20.sol";
import "./interface/EIP20NonStandardInterface.sol";
import "./interface/InterestRateModel.sol";
import "./Comptroller.sol";
import "./RiskController.sol";

/**
  * @title Compound's CToken Contract
  * @notice Abstract base for CTokens
  * @author Compound
  */
contract CToken is CTokenInterface, Exponential, TokenErrorReporter {

        /**
          * @notice Initialize the money market
          * @param comptroller_ The address of the Comptroller
          * @param interestRateModel_ The address of the interest rate model
          * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
```

```
     * @param name_ EIP-20 name of this token
     * @param symbol_ EIP-20 symbol of this token
     * @param decimals_ EIP-20 decimal precision of this token
     */
    function initialize(ComptrollerInterface comptroller_,
                        InterestRateModel interestRateModel_,
                        uint initialExchangeRateMantissa_,
                        string memory name_,
                        string memory symbol_,
                        uint8 decimals_) public {
        require(msg.sender == admin, "only admin may initialize the market");
        require(accrualBlockNumber == 0 && borrowIndex == 0, "market may only be initialized once");

        // Set initial exchange rate
        initialExchangeRateMantissa = initialExchangeRateMantissa_;
        require(initialExchangeRateMantissa > 0, "initial exchange rate must be greater than zero.");

        // Set the comptroller
        uint err = _setComptroller(comptroller_);
        require(err == uint(Error.NO_ERROR), "setting comptroller failed");

        // Initialize block number and borrow index (block number mocks depend on comptroller being set)
        accrualBlockNumber = getBlockNumber();
        borrowIndex = mantissaOne;

        // Set the interest rate model (depends on block number / borrow index)
        err = _setInterestRateModelFresh(interestRateModel_);
        require(err == uint(Error.NO_ERROR), "setting interest rate model failed");

        name = name_;
        symbol = symbol_;
        decimals = decimals_;

        // The counter starts true to prevent changing it from zero to non-zero (i.e. smaller cost/refund)
        _notEntered = true;
    }

    /**
     * @notice Transfer `tokens` tokens from `src` to `dst` by `spender`
     * @dev Called by both `transfer` and `transferFrom` internally
     * @param spender The address of the account performing the transfer
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param tokens The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferTokens(address spender, address src, address dst, uint tokens) internal returns (uint) {
        /* Fail if transfer not allowed */
        uint allowed = comptroller.transferAllowed(address(this), src, dst, tokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.TRANSFER_COMPTROLLER_REJECTION, allowed);
        }

        /* Do not allow self-transfers */
        if (src == dst) {
            return fail(Error.BAD_INPUT, FailureInfo.TRANSFER_NOT_ALLOWED);
        }

        /* Get the allowance, infinite for the account owner */
        uint startingAllowance = 0;
        if (spender == src) {
            startingAllowance = uint(-1);
        } else {
            startingAllowance = transferAllowances[src][spender];
        }

        /* Do the calculations, checking for {under,over}flow */
        MathError mathErr;
        uint allowanceNew;
        uint srcTokensNew;
        uint dstTokensNew;

        (mathErr, allowanceNew) = subUInt(startingAllowance, tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ALLOWED);
        }

        (mathErr, srcTokensNew) = subUInt(accountTokens[src], tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ENOUGH);
        }
```

```
        (mathErr, dstTokensNew) = addUInt(accountTokens[dst], tokens);
        if (mathErr != MathError.NO_ERROR) {
            return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_TOO_MUCH);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        accountTokens[src] = srcTokensNew;
        accountTokens[dst] = dstTokensNew;

        /* Eat some of the allowance (if necessary) */
        if (startingAllowance != uint(-1)) {
            transferAllowances[src][spender] = allowanceNew;
        }

        /* We emit a Transfer event */
        emit Transfer(src, dst, tokens);

        comptroller.transferVerify(address(this), src, dst, tokens);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 amount) external nonReentrant returns (bool) {
        return transferTokens(msg.sender, msg.sender, dst, amount) == uint(Error.NO_ERROR);
    }

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(address src, address dst, uint256 amount) external nonReentrant returns
(bool) {
        return transferTokens(msg.sender, src, dst, amount) == uint(Error.NO_ERROR);
    }

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param amount The number of tokens that are approved (-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 amount) external returns (bool) {
        address src = msg.sender;
        transferAllowances[src][spender] = amount;
        emit Approval(src, spender, amount);
        return true;
    }

    /**
     * @notice Get the current allowance from `owner` for `spender`
     * @param owner The address of the account which owns the tokens to be spent
     * @param spender The address of the account which may transfer tokens
     * @return The number of tokens allowed to be spent (-1 means infinite)
     */
    function allowance(address owner, address spender) external view returns (uint256) {
        return transferAllowances[owner][spender];
    }

    /**
     * @notice Get the token balance of the `owner`
     * @param owner The address of the account to query
     * @return The number of tokens owned by `owner`
     */
    function balanceOf(address owner) external view returns (uint256) {
        return accountTokens[owner];
    }

    /**
     * @notice Get the underlying balance of the `owner`
     * @dev This also accrues interest in a transaction
     * @param owner The address of the account to query
     * @return The amount of underlying owned by `owner`
```

```
     */
    function balanceOfUnderlying(address owner) external returns (uint) {
        Exp memory exchangeRate = Exp({mantissa : exchangeRateCurrent()});
        (MathError mErr, uint balance) = mulScalarTruncate(exchangeRate, accountTokens[owner]);
        require(mErr == MathError.NO_ERROR, "balance could not be calculated");
        return balance;
    }

    function balanceOfUnderlyingView(address owner) external view returns (uint){
        Exp memory exchangeRate = Exp({mantissa : exchangeRateStored()});
        (MathError mErr, uint balance) = mulScalarTruncate(exchangeRate, accountTokens[owner]);
        require(mErr == MathError.NO_ERROR, "balance could not be calculated");
        return balance;
    }

    /**
     * @notice Get a snapshot of the account's balances, and the cached exchange rate
     * @dev This is used by comptroller to more efficiently perform liquidity checks.
     * @param account Address of the account to snapshot
     * @return (possible error, token balance, borrow balance, exchange rate mantissa)
     */
    function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint) {
        uint cTokenBalance = accountTokens[account];
        uint borrowBalance;
        uint exchangeRateMantissa;

        MathError mErr;

        (mErr, borrowBalance) = borrowBalanceStoredInternal(account);
        if (mErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0, 0, 0);
        }

        (mErr, exchangeRateMantissa) = exchangeRateStoredInternal();
        if (mErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0, 0, 0);
        }

        return (uint(Error.NO_ERROR), cTokenBalance, borrowBalance, exchangeRateMantissa);
    }

    /**
     * @dev Function to simply retrieve block number
     *  This exists mainly for inheriting test contracts to stub this result.
     */
    function getBlockNumber() internal view returns (uint) {
        return block.number;
    }

    /**
     * @notice Returns the current per-block borrow interest rate for this cToken
     * @return The borrow interest rate per block, scaled by 1e18
     */
    function borrowRatePerBlock() external view returns (uint) {
        return interestRateModel.getBorrowRate(getCashPrior(), totalBorrows, totalReserves);
    }

    /**
     * @notice Returns the current per-block supply interest rate for this cToken
     * @return The supply interest rate per block, scaled by 1e18
     */
    function supplyRatePerBlock() external view returns (uint) {
        return interestRateModel.getSupplyRate(getCashPrior(), totalBorrows, totalReserves, reserveFactorMantissa);
    }

    /**
     * @notice Returns the current total borrows plus accrued interest
     * @return The total borrows with interest
     */
    function totalBorrowsCurrent() external nonReentrant returns (uint) {
        require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
        return totalBorrows;
    }

    /**
     * @notice Accrue interest to updated borrowIndex and then calculate
     *  account's borrow balance using the updated borrowIndex
     * @param account The address whose balance should be calculated after updating borrowIndex
     * @return The calculated balance
     */
    function borrowBalanceCurrent(address account) external nonReentrant returns (uint) {
        require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
        return borrowBalanceStored(account);
    }
```

```
    /**
     * @notice Return the borrow balance of account based on stored data
     * @param account The address whose balance should be calculated
     * @return The calculated balance
     */
    function borrowBalanceStored(address account) public view returns (uint) {
        (MathError err, uint result) = borrowBalanceStoredInternal(account);
        require(err == MathError.NO_ERROR, "borrowBalanceStored: borrowBalanceStoredInternal failed");
        return result;
    }

    /**
     * @notice Return the borrow balance of account based on stored data
     * @param account The address whose balance should be calculated
     * @return (error code, the calculated balance or 0 if error code is non-zero)
     */
    function borrowBalanceStoredInternal(address account) internal view returns (MathError, uint) {
        /* Note: we do not assert that the market is up to date */
        MathError mathErr;
        uint principalTimesIndex;
        uint result;

        /* Get borrowBalance and borrowIndex */
        BorrowSnapshot storage borrowSnapshot = accountBorrows[account];

        /* If borrowBalance = 0 then borrowIndex is likely also 0.
         * Rather than failing the calculation with a division by 0, we immediately return 0 in this case.
         */
        if (borrowSnapshot.principal == 0) {
            return (MathError.NO_ERROR, 0);
        }

        /* Calculate new borrow balance using the interest index:
         *  recentBorrowBalance = borrower.borrowBalance * market.borrowIndex / borrower.borrowIndex
         */
        (mathErr, principalTimesIndex) = mulUInt(borrowSnapshot.principal, borrowIndex);
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        (mathErr, result) = divUInt(principalTimesIndex, borrowSnapshot.interestIndex);
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        return (MathError.NO_ERROR, result);
    }

    /**
     * @notice Accrue interest then return the up-to-date exchange rate
     * @return Calculated exchange rate scaled by 1e18
     */
    function exchangeRateCurrent() public nonReentrant returns (uint) {
        require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
        return exchangeRateStored();
    }

    /**
     * @notice Calculates the exchange rate from the underlying to the CToken
     * @dev This function does not accrue interest before calculating the exchange rate
     * @return Calculated exchange rate scaled by 1e18
     */
    function exchangeRateStored() public view returns (uint) {
        (MathError err, uint result) = exchangeRateStoredInternal();
        require(err == MathError.NO_ERROR, "exchangeRateStored: exchangeRateStoredInternal failed");
        return result;
    }

    /**
     * @notice Calculates the exchange rate from the underlying to the CToken
     * @dev This function does not accrue interest before calculating the exchange rate
     * @return (error code, calculated exchange rate scaled by 1e18)
     */
    function exchangeRateStoredInternal() internal view returns (MathError, uint) {
        uint _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            /*
             * If there are no tokens minted:
             *  exchangeRate = initialExchangeRate
             */
            return (MathError.NO_ERROR, initialExchangeRateMantissa);
        } else {
```

```
                /*
                 * Otherwise:
                 *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
                 */
                uint totalCash = getCashPrior();
                uint cashPlusBorrowsMinusReserves;
                Exp memory exchangeRate;
                MathError mathErr;

                (mathErr, cashPlusBorrowsMinusReserves) = addThenSubUInt(totalCash, totalBorrows,
totalReserves);
                if (mathErr != MathError.NO_ERROR) {
                    return (mathErr, 0);
                }

                (mathErr, exchangeRate) = getExp(cashPlusBorrowsMinusReserves, _totalSupply);
                if (mathErr != MathError.NO_ERROR) {
                    return (mathErr, 0);
                }

                return (MathError.NO_ERROR, exchangeRate.mantissa);
            }
        }

        /**
         * @notice Get cash balance of this cToken in the underlying asset
         * @return The quantity of underlying asset owned by this contract
         */
        function getCash() external view returns (uint) {
            return getCashPrior();
        }

        /**
         * @notice Applies accrued interest to total borrows and reserves
         * @dev This calculates interest accrued from the last checkpointed block
         *   up to the current block and writes new checkpoint to storage.
         */
        function accrueInterest() public returns (uint) {
            /* Remember the initial block number */
            uint currentBlockNumber = getBlockNumber();
            uint accrualBlockNumberPrior = accrualBlockNumber;

            /* Short-circuit accumulating 0 interest */
            if (accrualBlockNumberPrior == currentBlockNumber) {
                return uint(Error.NO_ERROR);
            }

            /* Read the previous values out of storage */
            uint cashPrior = getCashPrior();
            uint borrowsPrior = totalBorrows;
            uint reservesPrior = totalReserves;
            uint borrowIndexPrior = borrowIndex;

            /* Calculate the current borrow interest rate */
            uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior,
reservesPrior);
            require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");

            /* Calculate the number of blocks elapsed since the last accrual */
            (MathError mathErr, uint blockDelta) = subUInt(currentBlockNumber,
accrualBlockNumberPrior);
            require(mathErr == MathError.NO_ERROR, "could not calculate block delta");

            /*
              * Calculate the interest accumulated into borrows and reserves and the new index:
              *  simpleInterestFactor = borrowRate * blockDelta
              *  interestAccumulated = simpleInterestFactor * totalBorrows
              *  totalBorrowsNew = interestAccumulated + totalBorrows
              *  totalReservesNew = interestAccumulated * reserveFactor + totalReserves
              *  borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
              */

            Exp memory simpleInterestFactor;
            uint interestAccumulated;
            uint totalBorrowsNew;
            uint totalReservesNew;
            uint borrowIndexNew;

            (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa : borrowRateMantissa}),
blockDelta);
            if (mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,

FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
                    uint(mathErr)
                );
```

```
        }

        (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(
            Exp({mantissa : reserveFactorMantissa}),
            interestAccumulated,
            reservesPrior
        );
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor,
borrowIndexPrior, borrowIndexPrior);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
                uint(mathErr)
            );
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We write the previously calculated values into storage */
        accrualBlockNumber = currentBlockNumber;
        borrowIndex = borrowIndexNew;
        totalBorrows = totalBorrowsNew;
        totalReserves = totalReservesNew;

        /* We emit an AccrueInterest event */
        emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sender supplies assets into the market and receives cTokens in exchange
      * @dev Accrues interest whether or not the operation succeeds, unless reverted
      * @param mintAmount The amount of the underlying asset to supply
      * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
      *                          and the actual mint amount.
      */
    function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
            return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
        }
        // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need to
        return mintFresh(msg.sender, mintAmount);
    }

    struct MintLocalVars {
```

```
        Error err;
        MathError mathErr;
        uint exchangeRateMantissa;
        uint mintTokens;
        uint totalSupplyNew;
        uint accountTokensNew;
        uint actualMintAmount;
    }

    /**
     * @notice User supplies assets into the market and receives cTokens in exchange
     * @dev Assumes interest has already been accrued up to the current block
     * @param minter The address of the account which is supplying the assets
     * @param mintAmount The amount of the underlying asset to supply
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
     *             and the actual mint amount.
     */
    function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
        /* Fail if mint not allowed */
        uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
        }

        MintLocalVars memory vars;

        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr)), 0);
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
          *  We call `doTransferIn` for the minter and the mintAmount.
          *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
          *  `doTransferIn` reverts if anything goes wrong, since we can't be sure if
          *  side-effects occurred. The function returns the amount actually transferred,
          *  in case of a fee. On success, the cToken holds an additional `actualMintAmount`
          *  of cash.
          */
        vars.actualMintAmount = doTransferIn(minter, mintAmount);

        /*
         * We get the current exchange rate and calculate the number of cTokens to be minted:
         *  mintTokens = actualMintAmount / exchangeRate
         */

        (vars.mathErr, vars.mintTokens) = divScalarByExpTruncate(
            vars.actualMintAmount,
            Exp({mantissa : vars.exchangeRateMantissa})
        );
        require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");

        /*
         * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
         *  totalSupplyNew = totalSupply + mintTokens
         *  accountTokensNew = accountTokens[minter] + mintTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens);
        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");

        (vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter], vars.mintTokens);
        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[minter] = vars.accountTokensNew;

        /* We emit a Mint event, and a Transfer event */
        emit Mint(minter, vars.actualMintAmount, vars.mintTokens);
        emit Transfer(address(this), minter, vars.mintTokens);
```

```
        /* We call the defense hook */
        comptroller.mintVerify(address(this), minter, vars.actualMintAmount, vars.mintTokens);

        return (uint(Error.NO_ERROR), vars.actualMintAmount);
    }

    /**
      * @notice Sender redeems cTokens in exchange for the underlying asset
      * @dev Accrues interest whether or not the operation succeeds, unless reverted
      * @param redeemTokens The number of cTokens to redeem into underlying
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function redeemInternal(uint redeemTokens) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem failed
            return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
        }
        // redeemFresh emits redeem-specific logs on errors, so we don't need to
        return redeemFresh(msg.sender, redeemTokens, 0);
    }

    /**
      * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
      * @dev Accrues interest whether or not the operation succeeds, unless reverted
      * @param redeemAmount The amount of underlying to receive from redeeming cTokens
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function redeemUnderlyingInternal(uint redeemAmount) internal nonReentrant returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem failed
            return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
        }
        // redeemFresh emits redeem-specific logs on errors, so we don't need to
        return redeemFresh(msg.sender, 0, redeemAmount);
    }

    struct RedeemLocalVars {
        Error err;
        MathError mathErr;
        uint exchangeRateMantissa;
        uint redeemTokens;
        uint redeemAmount;
        uint totalSupplyNew;
        uint accountTokensNew;
    }

    /**
      * @notice User redeems cTokens in exchange for the underlying asset
      * @dev Assumes interest has already been accrued up to the current block
      * @param redeemer The address of the account which is redeeming the tokens
      * @param redeemTokensIn The number of cTokens to redeem into underlying
      *        (only one of redeemTokensIn or redeemAmountIn may be non-zero)
      * @param redeemAmountIn The number of underlying tokens to receive from redeeming cTokens
      *        (only one of redeemTokensIn or redeemAmountIn may be non-zero)
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal returns (uint) {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");

        RedeemLocalVars memory vars;

        /* exchangeRate = invoke Exchange Rate Stored() */
        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr));
        }

        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
              * We calculate the exchange rate and the amount of underlying to be redeemed:
              *    redeemTokens = redeemTokensIn
              *    redeemAmount = redeemTokensIn x exchangeRateCurrent
              */
            vars.redeemTokens = redeemTokensIn;

            (vars.mathErr, vars.redeemAmount) = mulScalarTruncate(
                Exp({mantissa : vars.exchangeRateMantissa}), redeemTokensIn
            );
```

```
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */

            (vars.mathErr, vars.redeemTokens) = divScalarByExpTruncate(
                redeemAmountIn,
                Exp({mantissa : vars.exchangeRateMantissa})
            );
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            vars.redeemAmount = redeemAmountIn;
        }

        /* Fail if redeem not allowed */
        uint allowed = comptroller.redeemAllowed(address(this), redeemer, vars.redeemTokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.REDEEM_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH,
FailureInfo.REDEEM_FRESHNESS_CHECK);
        }

        /*
         * We calculate the new total supply and redeemer balance, checking for underflow:
         *  totalSupplyNew = totalSupply - redeemTokens
         *  accountTokensNew = accountTokens[redeemer] - redeemTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = subUInt(totalSupply, vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

        (vars.mathErr, vars.accountTokensNew) = subUInt(accountTokens[redeemer],
vars.redeemTokens);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

        /* Fail gracefully if protocol has insufficient cash */
        if (getCashPrior() < vars.redeemAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.REDEEM_TRANSFER_OUT_NOT_POSSIBLE);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects
occurred.
         */
        uint _amount = vars.redeemAmount;
        uint _feeAmount = div_(mul_(_amount, redeemFee), expScale);
        totalReserves = add_(totalReserves, _feeAmount);
        _amount = sub_(_amount, _feeAmount);
```

```
        doTransferOut(redeemer, vars.redeemAmount);

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[redeemer] = vars.accountTokensNew;

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), vars.redeemTokens);
        emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

        /* We call the defense hook */
        comptroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.redeemTokens);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sender borrows assets from the protocol to their own address
      * @param borrowAmount The amount of the underlying asset to borrow
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
        address riskControl = Comptroller(address(comptroller)).getRiskControl();
        if (riskControl != address(0)) {
            uint risk = RiskController(riskControl).checkMintRisk(address(this), msg.sender);
            require(risk == 0, 'risk control');
        }
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
            return fail(Error(error), FailureInfo.BORROW_ACCRUE_INTEREST_FAILED);
        }
        // borrowFresh emits borrow-specific logs on errors, so we don't need to
        return borrowFresh(msg.sender, borrowAmount);
    }

    struct BorrowLocalVars {
        MathError mathErr;
        uint accountBorrows;
        uint accountBorrowsNew;
        uint totalBorrowsNew;
    }

    /**
      * @notice Users borrow assets from the protocol to their own address
      * @param borrowAmount The amount of the underlying asset to borrow
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
        if (allowed != 0) {
            return                                  failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return                                  fail(Error.MARKET_NOT_FRESH,
FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (getCashPrior() < borrowAmount) {
            return                                  fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
          * We calculate the new borrower and total borrow balances, failing on overflow:
          *    accountBorrowsNew = accountBorrows + borrowAmount
          *    totalBorrowsNew = totalBorrows + borrowAmount
          */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(
                Error.MATH_ERROR,
                FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
                uint(vars.mathErr)
            );
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
        if (vars.mathErr != MathError.NO_ERROR) {
```

```
                return failOpaque(
                    Error.MATH_ERROR,
FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                );
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)

            /*
              * We invoke doTransferOut for the borrower and the borrowAmount.
              *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
              *  On success, the cToken borrowAmount less of cash.
              *   doTransferOut reverts if anything goes wrong, since we can't be sure if side effects
occurred.
              */
            doTransferOut(borrower, borrowAmount);

            /* We write the previously calculated values into storage */
            accountBorrows[borrower].principal = vars.accountBorrowsNew;
            accountBorrows[borrower].interestIndex = borrowIndex;
            totalBorrows = vars.totalBorrowsNew;

            /* We emit a Borrow event */
            emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

            /* We call the defense hook */
            comptroller.borrowVerify(address(this), borrower, borrowAmount);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Sender repays their own borrow
          * @param repayAmount The amount to repay
          * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
          *             and the actual repayment amount.
          */
        function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
                return                                                    (fail(Error(error),
FailureInfo.REPAY_BORROW_ACCRUE_INTEREST_FAILED), 0);
            }
            // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
            return repayBorrowFresh(msg.sender, msg.sender, repayAmount);
        }

        /**
          * @notice Sender repays a borrow belonging to borrower
          * @param borrower the account with the debt being payed off
          * @param repayAmount The amount to repay
          * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
          *     and the actual repayment amount.
          */
        function repayBorrowBehalfInternal(address borrower, uint repayAmount) internal nonReentrant
returns (uint, uint) {
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors, but we still want to log the fact that an attempted
borrow failed
                return                                                    (fail(Error(error),
FailureInfo.REPAY_BEHALF_ACCRUE_INTEREST_FAILED), 0);
            }
            // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
            return repayBorrowFresh(msg.sender, borrower, repayAmount);
        }

        struct RepayBorrowLocalVars {
            Error err;
            MathError mathErr;
            uint repayAmount;
            uint borrowerIndex;
```

```
            uint accountBorrows;
            uint accountBorrowsNew;
            uint totalBorrowsNew;
            uint actualRepayAmount;
        }

        /**
          * @notice Borrows are repaid by another user (possibly the borrower).
          * @param payer the account paying off the borrow
          * @param borrower the account with the debt being payed off
          * @param repayAmount the amount of undelrying tokens being returned
          * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
          *    and the actual repayment amount.
          */
        function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns (uint, uint) {
            /* Fail if repayBorrow not allowed */
            uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower, repayAmount);
            if (allowed != 0) {
                return (
                failOpaque(
                    Error.COMPTROLLER_REJECTION,
                    FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION,
                    allowed
                ), 0);
            }

            /* Verify market's block number equals current block number */
            if (accrualBlockNumber != getBlockNumber()) {
                return (fail(Error.MARKET_NOT_FRESH, FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0);
            }

            RepayBorrowLocalVars memory vars;

            /* We remember the original borrowerIndex for verification purposes */
            vars.borrowerIndex = accountBorrows[borrower].interestIndex;

            /* We fetch the amount the borrower owes, with accumulated interest */
            (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
            if (vars.mathErr != MathError.NO_ERROR) {
                return (failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
                    uint(vars.mathErr)
                ), 0);
            }

            /* If repayAmount == -1, repayAmount = accountBorrows */
            if (repayAmount == uint(-1)) {
                vars.repayAmount = vars.accountBorrows;
            } else {
                vars.repayAmount = repayAmount;
            }

            //////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)

            /*
              * We call doTransferIn for the payer and the repayAmount
              *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
              *  On success, the cToken holds an additional repayAmount of cash.
              *    doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
              *    it returns the amount actually transferred, in case of a fee.
              */
            vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

            /*
              * We calculate the new borrower and total borrow balances, failing on underflow:
              *  accountBorrowsNew = accountBorrows - actualRepayAmount
              *  totalBorrowsNew = totalBorrows - actualRepayAmount
              */
            (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.actualRepayAmount);
            require(vars.mathErr == MathError.NO_ERROR, "REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

            (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.actualRepayAmount);
            require(vars.mathErr == MathError.NO_ERROR, "REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

            /* We write the previously calculated values into storage */
            accountBorrows[borrower].principal = vars.accountBorrowsNew;
```

```
            accountBorrows[borrower].interestIndex = borrowIndex;
            totalBorrows = vars.totalBorrowsNew;

            /* We emit a RepayBorrow event */
            emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew,
vars.totalBorrowsNew);

            /* We call the defense hook */
            comptroller.repayBorrowVerify(address(this), payer, borrower, vars.actualRepayAmount,
vars.borrowerIndex);

            return (uint(Error.NO_ERROR), vars.actualRepayAmount);
        }

        /**
          * @notice The sender liquidates the borrowers collateral.
          *  The collateral seized is transferred to the liquidator.
          * @param borrower The borrower of this cToken to be liquidated
          * @param cTokenCollateral The market in which to seize collateral from the borrower
          * @param repayAmount The amount of the underlying borrowed asset to repay
          * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
          *                  and the actual repayment amount.
          */

        function liquidateBorrowInternal(
            address borrower,
            uint repayAmount,
            CTokenInterface cTokenCollateral
        ) internal nonReentrant returns (uint, uint) {
            uint error = accrueInterest();

            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but we still want to log the fact that an attempted liquidation failed
                return                                                    (fail(Error(error),
FailureInfo.LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED), 0);
            }

            error = cTokenCollateral.accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but we still want to log the fact that an attempted liquidation failed
                return                                                    (fail(Error(error),
FailureInfo.LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED), 0);
            }
            // liquidateBorrowFresh emits borrow-specific logs on errors, so we don't need to
            return liquidateBorrowFresh(msg.sender, borrower, repayAmount, cTokenCollateral);
        }

        /**
          * @notice The liquidator liquidates the borrowers collateral.
          *  The collateral seized is transferred to the liquidator.
          * @param borrower The borrower of this cToken to be liquidated
          * @param liquidator The address repaying the borrow and seizing collateral
          * @param cTokenCollateral The market in which to seize collateral from the borrower
          * @param repayAmount The amount of the underlying borrowed asset to repay
          * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol),
          *                  and the actual repayment amount.
          */
        function liquidateBorrowFresh(
            address liquidator,
            address borrower,
            uint repayAmount,
            CTokenInterface cTokenCollateral
        ) internal returns (uint, uint) {
            /* Fail if liquidate not allowed */
            uint allowed = comptroller.liquidateBorrowAllowed(
                address(this),
                address(cTokenCollateral),
                liquidator,
                borrower,
                repayAmount
            );
            if (allowed != 0) {
                return                                  (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
            }
            /* Verify market's block number equals current block number */
            if (accrualBlockNumber != getBlockNumber()) {
                return                                                    (fail(Error.MARKET_NOT_FRESH,
FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
            }

            /* Verify cTokenCollateral market's block number equals current block number */
            if (cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
                return                                                    (fail(Error.MARKET_NOT_FRESH,
FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK), 0);
```

```
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return (fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER), 0);
        }

        /* Fail if repayAmount = 0 */
        if (repayAmount == 0) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_ZERO), 0);
        }

        /* Fail if repayAmount = -1 */
        if (repayAmount == uint(- 1)) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
        }

        /* Fail if repayBorrow fails */
        (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower, repayAmount);
        if (repayBorrowError != uint(Error.NO_ERROR)) {
            return (fail(Error(repayBorrowError), FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED), 0);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We calculate the number of collateral tokens that will be seized */
        (uint amountSeizeError, uint seizeTokens) = comptroller.liquidateCalculateSeizeTokens(
            address(this),
            address(cTokenCollateral),
            actualRepayAmount
        );
        require(amountSeizeError == uint(Error.NO_ERROR), "LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");

        /* Revert if borrower collateral token balance < seizeTokens */
        require(cTokenCollateral.balanceOf(borrower) >= seizeTokens, "LIQUIDATE_SEIZE_TOO_MUCH");

        // If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make an external call
        uint seizeError;
        if (address(cTokenCollateral) == address(this)) {
            seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
        } else {
            seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
        }

        /* Revert if seize tokens fails (since we cannot be sure of side effects) */
        require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

        /* We emit a LiquidateBorrow event */
        emit LiquidateBorrow(liquidator, borrower, actualRepayAmount, address(cTokenCollateral), seizeTokens);

        /* We call the defense hook */
        comptroller.liquidateBorrowVerify(
            address(this),
            address(cTokenCollateral),
            liquidator,
            borrower,
            actualRepayAmount,
            seizeTokens
        );

        return (uint(Error.NO_ERROR), actualRepayAmount);
    }

    /**
     * @notice Transfers collateral tokens (this market) to the liquidator.
     * @dev Will fail unless called by another cToken during the process of liquidation.
     *  Its absolutely critical to use msg.sender as the borrowed cToken and not a parameter.
     * @param liquidator The account receiving seized collateral
     * @param borrower The account having collateral seized
     * @param seizeTokens The number of cTokens to seize
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function seize(address liquidator, address borrower, uint seizeTokens) external nonReentrant returns (uint) {
        return seizeInternal(msg.sender, liquidator, borrower, seizeTokens);
    }
```

```
    /**
      * @notice Transfers collateral tokens (this market) to the liquidator.
      * @dev Called only during an in-kind liquidation, or by liquidateBorrow during the liquidation of another CToken.
      *  Its absolutely critical to use msg.sender as the seizer cToken and not a parameter.
      * @param seizerToken The contract seizing the collateral (i.e. borrowed cToken)
      * @param liquidator The account receiving seized collateral
      * @param borrower The account having collateral seized
      * @param seizeTokens The number of cTokens to seize
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function seizeInternal(
        address seizerToken,
        address liquidator,
        address borrower,
        uint seizeTokens
    ) internal returns (uint) {
        /* Fail if seize not allowed */
        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, seizeTokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER);
        }

        MathError mathErr;
        uint borrowerTokensNew;
        uint liquidatorTokensNew;

        /*
          * We calculate the new borrower and liquidator token balances, failing on underflow/overflow:
          *  borrowerTokensNew = accountTokens[borrower] - seizeTokens
          *  liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
          */
        (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED, uint(mathErr));
        }

        (mathErr, liquidatorTokensNew) = addUInt(accountTokens[liquidator], seizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We write the previously calculated values into storage */
        accountTokens[borrower] = borrowerTokensNew;
        accountTokens[liquidator] = liquidatorTokensNew;

        /* Emit a Transfer event */
        emit Transfer(borrower, liquidator, seizeTokens);

        /* We call the defense hook */
        comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower, seizeTokens);

        return uint(Error.NO_ERROR);
    }


    /*** Admin Functions ***/

    /**
      * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
      * @dev Admin function to begin change of admin.
      *  The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
      * @param newPendingAdmin New pending admin.
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPendingAdmin(address payable newPendingAdmin) external returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
```

```
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
      * @dev Admin function for pending admin to accept role and update admin
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _acceptAdmin() external returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }

        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets a new comptroller for the market
      * @dev Admin function to set a new comptroller
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setComptroller(ComptrollerInterface newComptroller) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COMPTROLLER_OWNER_CHECK);
        }

        ComptrollerInterface oldComptroller = comptroller;
        // Ensure invoke comptroller.isComptroller() returns true
        require(newComptroller.isComptroller(), "marker method returned false");

        // Set market's comptroller to newComptroller
        comptroller = newComptroller;

        // Emit NewComptroller(oldComptroller, newComptroller)
        emit NewComptroller(oldComptroller, newComptroller);

        return uint(Error.NO_ERROR);
    }

    function _setFeeManager(address payable _feeManager) external returns (uint){
        require(msg.sender == admin, "!admin");
        feeManager = _feeManager;
        return 0;
    }

    function _setRedeemFactorMantissa(uint _redeemFee) external {
        require(msg.sender == admin, "!admin");
        require(_redeemFee > 0 && _redeemFee < 1e16, "beyond the scope");
        redeemFee = _redeemFee;

    }

    /**
      * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
      * @dev Admin function to accrue interest and set a new reserve factor
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setReserveFactor(uint newReserveFactorMantissa) external nonReentrant returns (uint)
```

```
{
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but on top of that we want to log the fact that an attempted reserve factor change
failed.
                return                                                    fail(Error(error),
FailureInfo.SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED);
            }
            // _setReserveFactorFresh emits reserve-factor-specific logs on errors, so we don't need to.
            return _setReserveFactorFresh(newReserveFactorMantissa);
        }

        /**
          * @notice Sets a new reserve factor for the protocol (*requires fresh interest accrual)
          * @dev Admin function to set a new reserve factor
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _setReserveFactorFresh(uint newReserveFactorMantissa) internal returns (uint) {
            // Check caller is admin
            if (msg.sender != admin) {
                return                                                fail(Error.UNAUTHORIZED,
FailureInfo.SET_RESERVE_FACTOR_ADMIN_CHECK);
            }

            // Verify market's block number equals current block number
            if (accrualBlockNumber != getBlockNumber()) {
                return                                            fail(Error.MARKET_NOT_FRESH,
FailureInfo.SET_RESERVE_FACTOR_FRESH_CHECK);
            }

            // Check newReserveFactor ≤ maxReserveFactor
            if (newReserveFactorMantissa > reserveFactorMaxMantissa) {
                return                                                fail(Error.BAD_INPUT,
FailureInfo.SET_RESERVE_FACTOR_BOUNDS_CHECK);
            }

            uint oldReserveFactorMantissa = reserveFactorMantissa;
            reserveFactorMantissa = newReserveFactorMantissa;

            emit NewReserveFactor(oldReserveFactorMantissa, newReserveFactorMantissa);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Accrues interest and reduces reserves by transferring from msg.sender
          * @param addAmount Amount of addition to reserves
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _addReservesInternal(uint addAmount) internal nonReentrant returns (uint) {
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but on top of that we want to log the fact that an attempted reduce reserves failed.
                return                                                fail(Error(error),
FailureInfo.ADD_RESERVES_ACCRUE_INTEREST_FAILED);
            }

            // _addReservesFresh emits reserve-addition-specific logs on errors, so we don't need to.
            (error,) = _addReservesFresh(addAmount);
            return error;
        }

        /**
          * @notice Add reserves by transferring from caller
          * @dev Requires fresh interest accrual
          * @param addAmount Amount of addition to reserves
          * @return (uint, uint) An error code (0=success, otherwise a failure (see ErrorReporter.sol for
details))
          *     and the actual amount added, net token fees
          */
        function _addReservesFresh(uint addAmount) internal returns (uint, uint) {
            // totalReserves + actualAddAmount
            uint totalReservesNew;
            uint actualAddAmount;

            // We fail gracefully unless market's block number equals current block number
            if (accrualBlockNumber != getBlockNumber()) {
                return                                        (fail(Error.MARKET_NOT_FRESH,
FailureInfo.ADD_RESERVES_FRESH_CHECK), actualAddAmount);
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)
```

```
            /*
              * We call doTransferIn for the caller and the addAmount
              *   Note: The cToken must handle variations between ERC-20 and ETH underlying.
              *   On success, the cToken holds an additional addAmount of cash.
              *    doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
              *   it returns the amount actually transferred, in case of a fee.
              */

            actualAddAmount = doTransferIn(msg.sender, addAmount);

            totalReservesNew = totalReserves + actualAddAmount;

            /* Revert on overflow */
            require(totalReservesNew >= totalReserves, "add reserves unexpected overflow");

            // Store reserves[n+1] = reserves[n] + actualAddAmount
            totalReserves = totalReservesNew;

            /* Emit NewReserves(admin, actualAddAmount, reserves[n+1]) */
            emit ReservesAdded(msg.sender, actualAddAmount, totalReservesNew);

            /* Return (NO_ERROR, actualAddAmount) */
            return (uint(Error.NO_ERROR), actualAddAmount);
        }


        /**
          * @notice Accrues interest and reduces reserves by transferring to admin
          * @param reduceAmount Amount of reduction to reserves
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _reduceReserves(uint reduceAmount) external nonReentrant returns (uint) {
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                // accrueInterest emits logs on errors,
                // but on top of that we want to log the fact that an attempted reduce reserves failed.
                return fail(Error(error), FailureInfo.REDUCE_RESERVES_ACCRUE_INTEREST_FAILED);
            }
            // _reduceReservesFresh emits reserve-reduction-specific logs on errors, so we don't need to.
            return _reduceReservesFresh(reduceAmount);
        }

        /**
          * @notice Reduces reserves by transferring to admin
          * @dev Requires fresh interest accrual
          * @param reduceAmount Amount of reduction to reserves
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
            // totalReserves - reduceAmount
            uint totalReservesNew;

            // Check caller is admin
            if (msg.sender != admin) {
                return fail(Error.UNAUTHORIZED, FailureInfo.REDUCE_RESERVES_ADMIN_CHECK);
            }

            // We fail gracefully unless market's block number equals current block number
            if (accrualBlockNumber != getBlockNumber()) {
                return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDUCE_RESERVES_FRESH_CHECK);
            }

            // Fail gracefully if protocol has insufficient underlying cash
            if (getCashPrior() < reduceAmount) {
                return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.REDUCE_RESERVES_CASH_NOT_AVAILABLE);
            }

            // Check reduceAmount ≤ reserves[n] (totalReserves)
            if (reduceAmount > totalReserves) {
                return fail(Error.BAD_INPUT, FailureInfo.REDUCE_RESERVES_VALIDATION);
            }

            ////////////////////////
            // EFFECTS & INTERACTIONS
            // (No safe failures beyond this point)

            totalReservesNew = totalReserves - reduceAmount;
            // We checked reduceAmount <= totalReserves above, so this should never revert.
            require(totalReservesNew <= totalReserves, "reduce reserves unexpected underflow");

            // Store reserves[n+1] = reserves[n] - reduceAmount
            totalReserves = totalReservesNew;
```

```
        // doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
        doTransferOut(feeManager, reduceAmount);

        emit ReservesReduced(admin, reduceAmount, totalReservesNew);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice accrues interest and updates the interest rate model using _setInterestRateModelFresh
      * @dev Admin function to accrue interest and update the interest rate model
      * @param newInterestRateModel the new interest rate model to use
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors,
            // but on top of that we want to log the fact that an attempted change of interest rate model failed
            return fail(Error(error), FailureInfo.SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED);
        }
        // _setInterestRateModelFresh emits interest-rate-model-update-specific logs on errors, so we don't need to.
        return _setInterestRateModelFresh(newInterestRateModel);
    }

    /**
      * @notice updates the interest rate model (*requires fresh interest accrual)
      * @dev Admin function to update the interest rate model
      * @param newInterestRateModel the new interest rate model to use
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setInterestRateModelFresh(InterestRateModel newInterestRateModel) internal returns (uint) {

        // Used to store old model for use in the event that is emitted on success
        InterestRateModel oldInterestRateModel;

        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_INTEREST_RATE_MODEL_OWNER_CHECK);
        }

        // We fail gracefully unless market's block number equals current block number
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.SET_INTEREST_RATE_MODEL_FRESH_CHECK);
        }

        // Track the market's current interest rate model
        oldInterestRateModel = interestRateModel;

        // Ensure invoke newInterestRateModel.isInterestRateModel() returns true
        require(newInterestRateModel.isInterestRateModel(), "marker method returned false");

        // Set the interest rate model to newInterestRateModel
        interestRateModel = newInterestRateModel;

        // Emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel)
        emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel);

        return uint(Error.NO_ERROR);
    }

    /*** Safe Token ***/

    /**
      * @notice Gets balance of this contract in terms of the underlying
      * @dev This excludes the value of the current message, if any
      * @return The quantity of underlying owned by this contract
      */
    function getCashPrior() internal view returns (uint);

    /**
      * @dev Performs a transfer in, reverting upon failure.
      *  Returns the amount actually transferred to the protocol, in case of a fee.
      *  This may revert due to insufficient balance or insufficient allowance.
      */
    function doTransferIn(address from, uint amount) internal returns (uint);

    /**
```

```
        * @dev Performs a transfer out, ideally returning an explanatory error code upon failure tather than reverting.
        *  If caller has not called checked protocol's balance, may revert due to insufficient cash held in the contract.
        *  If caller has checked protocol's balance, and verified it is >= amount,
        *  this should not revert in normal conditions.
        */
       function doTransferOut(address payable to, uint amount) internal;


       /*** Reentrancy Guard ***/

       /**
        * @dev Prevents a contract from calling itself, directly or indirectly.
        */
       modifier nonReentrant() {
           require(_notEntered, "re-entered");
           _notEntered = false;
           _;
           _notEntered = true;
           // get a gas-refund post-Istanbul
       }


       /**
    * @dev allows smartcontracts to access the liquidity of the pool within one transaction,
    * as long as the amount taken plus a fee is returned.
                NOTE There are security concerns for developers of flashloan receiver contracts
    * that must be kept into consideration. For further details please visit https://developers.aave.com
    * @param _receiver The address of the contract receiving the funds.
                The receiver should implement the IFlashLoanReceiver interface.
    * @param _reserve the address of the principal reserve
    * @param _amount the amount requested for this flashloan
    **/
       function flashLoan(address _receiver, address _reserve, uint256 _amount, bytes memory _params)
       public nonReentrant returns (uint)
       {
           //check that the reserve has enough available liquidity
           //we avoid using the getAvailableLiquidity() function in LendingPoolCore to save gas
           uint256 availableLiquidityBefore = getCashPrior();

           require(
               availableLiquidityBefore >= _amount,
               "There is not enough liquidity available to borrow"
           );

           //calculate amount fee
           uint256 amountFee = div_(mul_(_amount, FLASHLOAN_FEE_TOTAL), 10000);

           require(
               amountFee > 0,
               "The requested amount is too small for a flashLoan."
           );

           //get the FlashLoanReceiver instance
           IFlashLoanReceiver receiver = IFlashLoanReceiver(_receiver);

           address payable userPayable = address(uint160(_receiver));

           //transfer funds to the receiver
           doTransferOut(userPayable, _amount);

           //execute action of the receiver
           receiver.executeOperation(_reserve, address(this), _amount, amountFee, _params);

           //check that the actual balance of the core contract includes the returned amount
           uint256 availableLiquidityAfter = getCashPrior();

           require(
               availableLiquidityAfter >= add_(availableLiquidityBefore, amountFee),
               "The actual balance of the protocol is inconsistent"
           );

           totalReserves = add_(totalReserves, amountFee);

           //solium-disable-next-line
           emit FlashLoan(_receiver, address(this), _reserve, _amount, amountFee, block.timestamp);
           return 0;
       }
}
```

ErrorReporter.sol

```
pragma solidity ^0.5.9;

contract ComptrollerErrorReporter {
```

```
    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        COMPTROLLER_MISMATCH,
        INSUFFICIENT_SHORTFALL,
        INSUFFICIENT_LIQUIDITY,
        INVALID_CLOSE_FACTOR,
        INVALID_COLLATERAL_FACTOR,
        INVALID_LIQUIDATION_INCENTIVE,
        MARKET_NOT_ENTERED, // no longer possible
        MARKET_NOT_LISTED,
        MARKET_ALREADY_LISTED,
        MATH_ERROR,
        NONZERO_BORROW_BALANCE,
        PRICE_ERROR,
        REJECTION,
        SNAPSHOT_ERROR,
        TOO_MANY_ASSETS,
        TOO_MUCH_REPAY
    }

    enum FailureInfo {
        ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
        ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK,
        EXIT_MARKET_BALANCE_OWED,
        EXIT_MARKET_REJECTION,
        SET_CLOSE_FACTOR_OWNER_CHECK,
        SET_CLOSE_FACTOR_VALIDATION,
        SET_COLLATERAL_FACTOR_OWNER_CHECK,
        SET_COLLATERAL_FACTOR_NO_EXISTS,
        SET_COLLATERAL_FACTOR_VALIDATION,
        SET_COLLATERAL_FACTOR_WITHOUT_PRICE,
        SET_IMPLEMENTATION_OWNER_CHECK,
        SET_LIQUIDATION_INCENTIVE_OWNER_CHECK,
        SET_LIQUIDATION_INCENTIVE_VALIDATION,
        SET_MAX_ASSETS_OWNER_CHECK,
        SET_PENDING_ADMIN_OWNER_CHECK,
        SET_PENDING_IMPLEMENTATION_OWNER_CHECK,
        SET_PRICE_ORACLE_OWNER_CHECK,
        SUPPORT_MARKET_EXISTS,
        SUPPORT_MARKET_OWNER_CHECK,
        SET_PAUSE_GUARDIAN_OWNER_CHECK
    }

    /**
      * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail` is an arbitrary
      * contract-specific code that enables us to report opaque error codes from upgradeable contracts.
      **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
      */
    function fail(Error err, FailureInfo info) internal returns (uint) {
        emit Failure(uint(err), uint(info), 0);

        return uint(err);
    }

    /**
      * @dev use this when reporting an opaque error from an upgradeable collaborator contract
      */
    function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
        emit Failure(uint(err), uint(info), opaqueError);

        return uint(err);
    }
}

contract TokenErrorReporter {

    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        BAD_INPUT,
        COMPTROLLER_REJECTION,
        COMPTROLLER_CALCULATION_ERROR,
        INTEREST_RATE_MODEL_ERROR,
        INVALID_ACCOUNT_PAIR,
        INVALID_CLOSE_AMOUNT_REQUESTED,
        INVALID_COLLATERAL_FACTOR,
        MATH_ERROR,
        MARKET_NOT_FRESH,
        MARKET_NOT_LISTED,
```

```
        TOKEN_INSUFFICIENT_ALLOWANCE,
        TOKEN_INSUFFICIENT_BALANCE,
        TOKEN_INSUFFICIENT_CASH,
        TOKEN_TRANSFER_IN_FAILED,
        TOKEN_TRANSFER_OUT_FAILED,
        RISK_CONTROL
    }

    /*
      * Note: FailureInfo (but not Error) is kept in alphabetical order
      *       This is because FailureInfo grows significantly faster, and
      *       the order of Error has some meaning, while the order of FailureInfo
      *       is entirely arbitrary.
      */
    enum FailureInfo {
        ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
        ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
        ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
        ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
        BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
        BORROW_ACCRUE_INTEREST_FAILED,
        BORROW_CASH_NOT_AVAILABLE,
        BORROW_FRESHNESS_CHECK,
        BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
        BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
        BORROW_MARKET_NOT_LISTED,
        BORROW_COMPTROLLER_REJECTION,
        LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED,
        LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED,
        LIQUIDATE_COLLATERAL_FRESHNESS_CHECK,
        LIQUIDATE_COMPTROLLER_REJECTION,
        LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED,
        LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX,
        LIQUIDATE_CLOSE_AMOUNT_IS_ZERO,
        LIQUIDATE_FRESHNESS_CHECK,
        LIQUIDATE_LIQUIDATOR_IS_BORROWER,
        LIQUIDATE_REPAY_BORROW_FRESH_FAILED,
        LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED,
        LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED,
        LIQUIDATE_SEIZE_COMPTROLLER_REJECTION,
        LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER,
        LIQUIDATE_SEIZE_TOO_MUCH,
        MINT_ACCRUE_INTEREST_FAILED,
        MINT_COMPTROLLER_REJECTION,
        MINT_EXCHANGE_CALCULATION_FAILED,
        MINT_EXCHANGE_RATE_READ_FAILED,
        MINT_FRESHNESS_CHECK,
        MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
        MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
        MINT_TRANSFER_IN_FAILED,
        MINT_TRANSFER_IN_NOT_POSSIBLE,
        REDEEM_ACCRUE_INTEREST_FAILED,
        REDEEM_COMPTROLLER_REJECTION,
        REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED,
        REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED,
        REDEEM_EXCHANGE_RATE_READ_FAILED,
        REDEEM_FRESHNESS_CHECK,
        REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED,
        REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED,
        REDEEM_TRANSFER_OUT_NOT_POSSIBLE,
        REDUCE_RESERVES_ACCRUE_INTEREST_FAILED,
        REDUCE_RESERVES_ADMIN_CHECK,
        REDUCE_RESERVES_CASH_NOT_AVAILABLE,
        REDUCE_RESERVES_FRESH_CHECK,
        REDUCE_RESERVES_VALIDATION,
        REPAY_BEHALF_ACCRUE_INTEREST_FAILED,
        REPAY_BORROW_ACCRUE_INTEREST_FAILED,
        REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_COMPTROLLER_REJECTION,
        REPAY_BORROW_FRESHNESS_CHECK,
        REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
        REPAY_BORROW_TRANSFER_IN_NOT_POSSIBLE,
        SET_COLLATERAL_FACTOR_OWNER_CHECK,
        SET_COLLATERAL_FACTOR_VALIDATION,
        SET_COMPTROLLER_OWNER_CHECK,
        SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED,
        SET_INTEREST_RATE_MODEL_FRESH_CHECK,
        SET_INTEREST_RATE_MODEL_OWNER_CHECK,
        SET_MAX_ASSETS_OWNER_CHECK,
        SET_ORACLE_MARKET_NOT_LISTED,
        SET_PENDING_ADMIN_OWNER_CHECK,
        SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED,
        SET_RESERVE_FACTOR_ADMIN_CHECK,
```

```
        SET_RESERVE_FACTOR_FRESH_CHECK,
        SET_RESERVE_FACTOR_BOUNDS_CHECK,
        TRANSFER_COMPTROLLER_REJECTION,
        TRANSFER_NOT_ALLOWED,
        TRANSFER_NOT_ENOUGH,
        TRANSFER_TOO_MUCH,
        ADD_RESERVES_ACCRUE_INTEREST_FAILED,
        ADD_RESERVES_FRESH_CHECK,
        ADD_RESERVES_TRANSFER_IN_NOT_POSSIBLE
    }

    /**
      * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail` is an arbitrary
      * contract-specific code that enables us to report opaque error codes from upgradeable contracts.
      **/
    event Failure(uint error, uint info, uint detail);

    /**
      * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
      */
    function fail(Error err, FailureInfo info) internal returns (uint) {
        emit Failure(uint(err), uint(info), 0);

        return uint(err);
    }

    /**
      * @dev use this when reporting an opaque error from an upgradeable collaborator contract
      */
    function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
        emit Failure(uint(err), uint(info), opaqueError);

        return uint(err);
    }
}
```

JumpRateModel.sol

```
pragma solidity ^0.5.9;

import "./interface/InterestRateModel.sol";
import "./lib/SafeMath.sol";

/**
  * @title Compound's JumpRateModel Contract
  * @author Compound
  */
contract JumpRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint jumpMultiplierPerBlock, uint kink);

    /**
      * @notice The approximate number of blocks per year that is assumed by the interest rate model
      */
    uint public constant blocksPerYear = 10512000;

    /**
      * @notice The multiplier of utilization rate that gives the slope of the interest rate
      */
    uint public multiplierPerBlock;

    /**
      * @notice The base interest rate which is the y-intercept when utilization rate is 0
      */
    uint public baseRatePerBlock;

    /**
      * @notice The multiplierPerBlock after hitting a specified utilization point
      */
    uint public jumpMultiplierPerBlock;

    /**
      * @notice The utilization point at which the jump multiplier is applied
      */
    uint public kink;

    /**
      * @notice Construct an interest rate model
      * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
      * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
      * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
```

```
        * @param kink_ The utilization point at which the jump multiplier is applied
        */
    constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_)
public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);
        jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
        kink = kink_;

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock,
kink);
    }

    /**
     * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market (currently unused)
     * @return The utilization rate as a mantissa between [0, 1e18]
     */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
     * @notice Calculates the current borrow rate per block, with the error code expected by the
market
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint util = utilizationRate(cash, borrows, reserves);

        if (util <= kink) {
            return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
        } else {
            uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
            uint excessUtil = util.sub(kink);
            return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
        }
    }

    /**
     * @notice Calculates the current supply rate per block
     * @param cash The amount of cash in the market
     * @param borrows The amount of borrows in the market
     * @param reserves The amount of reserves in the market
     * @param reserveFactorMantissa The current reserve factor for the market
     * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
     */
    function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```

JumpRateModelV2.sol

```
pragma solidity ^0.5.9;

import "./BaseJumpRateModelV2.sol";
import "./interface/InterestRateModel.sol";


/**
  * @title Compound's JumpRateModel Contract V2 for V2 cTokens
  * @author Arr00
  * @notice Supports only for V2 cTokens
  */
contract JumpRateModelV2 is InterestRateModel, BaseJumpRateModelV2 {

    /**
```

```
        * @notice Calculates the current borrow rate per block
        * @param cash The amount of cash in the market
        * @param borrows The amount of borrows in the market
        * @param reserves The amount of reserves in the market
        * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
        */
       function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint) {
            return getBorrowRateInternal(cash, borrows, reserves);
       }

       constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_,
address owner_)
       BaseJumpRateModelV2(baseRatePerYear,    multiplierPerYear,    jumpMultiplierPerYear,    kink_,
owner_) public {}
}
```

LegacyJumpRateModelV2.sol

```
pragma solidity ^0.5.9;

import "./BaseJumpRateModelV2.sol";
import "./interface/LegacyInterestRateModel.sol";


/**
   * @title Compound's JumpRateModel Contract V2 for legacy cTokens
   * @author Arr00
   * @notice Supports only legacy cTokens
   */
contract LegacyJumpRateModelV2 is LegacyInterestRateModel, BaseJumpRateModelV2 {

       /**
        * @notice Calculates the current borrow rate per block, with the error code expected by the
market
        * @param cash The amount of cash in the market
        * @param borrows The amount of borrows in the market
        * @param reserves The amount of reserves in the market
        * @return (Error, The borrow rate percentage per block as a mantissa (scaled by 1e18))
        */
       function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint, uint) {
            return (0, getBorrowRateInternal(cash, borrows, reserves));
       }

       constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_,
address owner_)
       BaseJumpRateModelV2(baseRatePerYear,    multiplierPerYear,    jumpMultiplierPerYear,    kink_,
owner_) public {}
}
```

Maximillion.sol

```
pragma solidity ^0.5.9;

import "./CNative.sol";

/**
  * @title Compound's Maximillion Contract
  * @author Compound
  */
contract Maximillion {
       /**
        * @notice The default cEther market to repay in
        */
       CNative public cEther;

       /**
        * @notice Construct a Maximillion to repay max in a CEther market
        */
       constructor(CNative cEther_) public {
            cEther = cEther_;
       }

       /**
        * @notice msg.sender sends Ether to repay an account's borrow in the cEther market
        * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
        * @param borrower The address of the borrower account to repay on behalf of
        */
       function repayBehalf(address borrower) public payable {
            repayBehalfExplicit(borrower, cEther);
       }

       /**
        * @notice msg.sender sends Ether to repay an account's borrow in a cEther market
        * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
        * @param borrower The address of the borrower account to repay on behalf of
```

```
     * @param cEther_ The address of the cEther contract to repay in
     */
    function repayBehalfExplicit(address borrower, CNative cEther_) public payable {
        uint received = msg.value;
        uint borrows = cEther_.borrowBalanceCurrent(borrower);
        if (received > borrows) {
            cEther_.repayBorrowBehalf.value(borrows)(borrower);
            msg.sender.transfer(received - borrows);
        } else {
            cEther_.repayBorrowBehalf.value(received)(borrower);
        }
    }
}
```

Migrations.sol

```
pragma solidity ^0.5.9;

contract Migrations {

}
```

RiskController.sol

```
pragma solidity ^0.5.9;

import "./lib/Controller.sol";

contract RiskController is Controller {

    uint constant   NO_RISK = 0;
    uint constant   HIGH = 1;


    mapping(address => bool) public contractWhiteList;

    mapping(address => bool) public banMint;
    mapping(address => bool) public banBorrow;

    function isContract(address _addr) public view returns (bool isContract){
        uint32 size;
        assembly {
            size := extcodesize(_addr)
        }
        return (size > 0);
    }


    function addToWhiteList(address _target) public onlyController {
        contractWhiteList[_target] = true;
    }

    function removeFromWhiteList(address _target) public onlyController {
        contractWhiteList[_target] = false;
    }

    function banTokenMint(address _token) public onlyController {
        banMint[_token] = true;
    }

    function removeBanTokenMint(address _token) public onlyController {
        banMint[_token] = false;
    }

    function banTokenBorrow(address _token) public onlyController {
        banBorrow[_token] = true;
    }

    function removeBanTokenBorrow(address _token) public onlyController {
        banBorrow[_token] = false;
    }

    function checkMintRisk(
        address _token,
        address _address
    ) external view returns (uint mintRisk) {
        if (!isContract(_address)) {
            return NO_RISK;
        }
        if (contractWhiteList[_address]) {
            return NO_RISK;
        }
        mintRisk = banMint[_token] ? HIGH : NO_RISK;
        return mintRisk;

    }

    function checkBorrowRisk(
```

```
            address _token,
            address _address
        ) external view returns (uint borrowRisk) {
            if (!isContract(_address)) {
                return NO_RISK;
            }
            if (contractWhiteList[_address]) {
                return NO_RISK;
            }
            borrowRisk = banBorrow[_token] ? HIGH : NO_RISK;
            return borrowRisk;
        }

}
```

Unitroller.sol

```
pragma solidity ^0.5.9;

import "./ErrorReporter.sol";
import "./interface/ComptrollerStorage.sol";
/**
  * @title ComptrollerCore
  * @dev Storage for the comptroller is at this address, while execution is delegated to the `comptrollerImplementation`.
  * CTokens should reference this contract as their comptroller.
  */
contract Unitroller is UnitrollerAdminStorage, ComptrollerErrorReporter {

    /**
      * @notice Emitted when pendingComptrollerImplementation is changed
      */
    event NewPendingImplementation(address oldPendingImplementation, address newPendingImplementation);

    /**
      * @notice Emitted when pendingComptrollerImplementation is accepted,
      *         which means comptroller implementation is updated
      */
    event NewImplementation(address oldImplementation, address newImplementation);

    /**
      * @notice Emitted when pendingAdmin is changed
      */
    event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

    /**
      * @notice Emitted when pendingAdmin is accepted, which means admin is updated
      */
    event NewAdmin(address oldAdmin, address newAdmin);

    constructor() public {
        // Set admin to caller
        admin = msg.sender;
    }

    /*** Admin Functions ***/
    function _setPendingImplementation(address newPendingImplementation) public returns (uint) {

        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_IMPLEMENTATION_OWNER_CHECK);
        }

        address oldPendingImplementation = pendingComptrollerImplementation;

        pendingComptrollerImplementation = newPendingImplementation;

        emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

        return uint(Error.NO_ERROR);
    }

    /**
    * @notice Accepts new implementation of comptroller. msg.sender must be pendingImplementation
    * @dev Admin function for new implementation to accept it's role as implementation
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
    function _acceptImplementation() public returns (uint) {
        // Check caller is pendingImplementation and pendingImplementation ≠ address(0)
        if (msg.sender != pendingComptrollerImplementation || pendingComptrollerImplementation == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK);
        }
```

```
            // Save current values for inclusion in log
            address oldImplementation = comptrollerImplementation;
            address oldPendingImplementation = pendingComptrollerImplementation;

            comptrollerImplementation = pendingComptrollerImplementation;

            pendingComptrollerImplementation = address(0);

            emit NewImplementation(oldImplementation, comptrollerImplementation);
            emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

            return uint(Error.NO_ERROR);
        }


        /**
          * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
          * @dev Admin function to begin change of admin.
          *      The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
          * @param newPendingAdmin New pending admin.
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _setPendingAdmin(address newPendingAdmin) public returns (uint) {
            // Check caller = admin
            if (msg.sender != admin) {
                return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
            }

            // Save current value, if any, for inclusion in log
            address oldPendingAdmin = pendingAdmin;

            // Store pendingAdmin with value newPendingAdmin
            pendingAdmin = newPendingAdmin;

            // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
            emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

            return uint(Error.NO_ERROR);
        }

        /**
          * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
          * @dev Admin function for pending admin to accept role and update admin
          * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
          */
        function _acceptAdmin() public returns (uint) {
            // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
            if (msg.sender != pendingAdmin || msg.sender == address(0)) {
                return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
            }

            // Save current values for inclusion in log
            address oldAdmin = admin;
            address oldPendingAdmin = pendingAdmin;

            // Store admin with value pendingAdmin
            admin = pendingAdmin;

            // Clear the pending value
            pendingAdmin = address(0);

            emit NewAdmin(oldAdmin, admin);
            emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

            return uint(Error.NO_ERROR);
        }

        /**
          * @dev Delegates execution to an implementation contract.
          * It returns to the external caller whatever the implementation returns
          * or forwards reverts.
          */
        function() payable external {
            // delegate all other functions to current implementation
            (bool success,) = comptrollerImplementation.delegatecall(msg.data);

            assembly {
                let free_mem_ptr := mload(0x40)
                returndatacopy(free_mem_ptr, 0, returndatasize)

                switch success
                case 0 {revert(free_mem_ptr, returndatasize)}
                default {return (free_mem_ptr, returndatasize)}
```

```
            }
        }
}
```

WhitePaperInterestRateModel.sol

```
pragma solidity ^0.5.9;

import "./interface/InterestRateModel.sol";
import "./lib/SafeMath.sol";

/**
  * @title Compound's WhitePaperInterestRateModel Contract
  * @author Compound
  * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper
  */
contract WhitePaperInterestRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock);

    /**
      * @notice The approximate number of blocks per year that is assumed by the interest rate model
      */
    uint public constant blocksPerYear = 10512000;

    /**
      * @notice The multiplier of utilization rate that gives the slope of the interest rate
      */
    uint public multiplierPerBlock;

    /**
      * @notice The base interest rate which is the y-intercept when utilization rate is 0
      */
    uint public baseRatePerBlock;

    /**
      * @notice Construct an interest rate model
      * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
      * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
      */
    constructor(uint baseRatePerYear, uint multiplierPerYear) public {
        baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
        multiplierPerBlock = multiplierPerYear.div(blocksPerYear);

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock);
    }

    /**
      * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market (currently unused)
      * @return The utilization rate as a mantissa between [0, 1e18]
      */
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }

    /**
      * @notice Calculates the current borrow rate per block, with the error code expected by the market
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market
      * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
      */
    function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
        uint ur = utilizationRate(cash, borrows, reserves);
        return ur.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
    }

    /**
      * @notice Calculates the current supply rate per block
      * @param cash The amount of cash in the market
      * @param borrows The amount of borrows in the market
      * @param reserves The amount of reserves in the market
      * @param reserveFactorMantissa The current reserve factor for the market
      * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
      */
```

```
    function getSupplyRate(
        uint cash,
        uint borrows,
        uint reserves,
        uint reserveFactorMantissa
    ) public view returns (uint) {
        uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
        uint borrowRate = getBorrowRate(cash, borrows, reserves);
        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }
}
```

6. Appendix B: Vulnerability risk rating criteria

Smart contract vulnerability rating standards	
Vulnerability rating	Vulnerability rating description
High-risk vulnerabilities	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.;
Mid-risk vulnerability	Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.;
Low-risk vulnerabilities	Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.

7. Appendix C: Introduction to vulnerability testing tools

7.1 Manticore

A Manticore is a symbolic execution tool for analyzing binary files and smart contracts. A Manticore consists of a symbolic Ethereum virtual machine (EVM), an EVM disassembler/assembler, and a convenient interface for automatic compilation and analysis of the Solarium body.It also incorporates Ethersplay, a Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis. Like binaries, Manticore provides a simple command-line interface and a Python API for analyzing EVM bytecode.

7.2 Oyente

Oyente is a smart contract analysis tool that can be used to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and so on.More conveniently, Oyente's design is modular, so this allows power users to implement and insert their own inspection logic to check the custom properties in their contracts.

7.3 securify. Sh

Securify verifies the security issues common to Ethereum's smart contracts, such as unpredictability of trades and lack of input verification, while fully automated and analyzing all possible execution paths, and Securify has a specific language for identifying vulnerabilities that enables the securities to focus on current security and other reliability issues at all times.

7.4 Echidna

Echidna is a Haskell library designed for fuzzy testing EVM code.

7.5 MAIAN

MAIAN is an automated tool used to find holes in Ethereum's smart contracts. MAIAN processes the bytecode of the contract and tries to set up a series of transactions to find and confirm errors.

7.6 ethersplay

Ethersplay is an EVM disassembler that includes correlation analysis tools.

7.7 IDA - evm entry

Ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

7.8 want - ide

Remix is a browser-based compiler and IDE that allows users to build ethereum contracts and debug transactions using Solarium language.

7.9 KnownSec Penetration Tester kit

KnownSec penetration tester's toolkit, developed, collected and used by KnownSec penetration tester engineers, contains batch automated testing tools, self-developed tools, scripts or utilization tools, etc. dedicated to testers.

Beijing Knownsec Information Tech.CO.,LTD

Telephone	+86(10)400 060 9587
Email	sec@knownsec.com
Official website	www.knownsec.com
Address	2509, block T2-B, WangJing SOHO, Chaoyang District, Beijing