Formal Verification Report: Silo v2 Protocol

- Competition: https://cantina.xyz/competitions/18f1e37b-9ac2-4ba9-b32e-50344500c1a7
- Repository: https://github.com/Certora/silo-v2-cantina-fv
- Latest Commit Hash: a45606c
- Scope: silo-core/contracts
- Date: January 2025
- Author: @alexzoid_eth
- Certora Prover version: 7.28.0

About Silo Protocol

Silo v2 is a decentralized lending protocol implementing isolated risk markets with a unique two-asset lending architecture. The protocol enables permissionless deployment of lending markets through the Silo Factory, creating two ERC-4626 vaults per market that are immutable by default with optional upgradeable settings.

About the Silo Architecture

The protocol features several innovative architectural components:

- **Collateral Mechanisms**: Two distinct collateral types - "borrowable deposits" that can earn interest and "non-borrowable deposits" that remain protected, supporting both partial and full liquidations
- **Token Architecture**: Three specialized token types including vault shares (representing borrowable deposits), protected collateral tokens, and share debt tokens (ERC20R)
- **Hooks System**: An extensible mechanism for interacting with core protocol actions, allowing custom logic integration
- **Liquidation Module**: Ensures market solvency through sophisticated liquidation mechanisms
- **Oracle System**: Flexible oracle integration supporting multiple price feed mechanisms (Chainlink, Uniswap V3, DIA) with 0-2 oracles per token asset and built-in fallback pricing
- **Dynamic Interest Rate Model**: Utilizes a PI controller with built-in safety mechanisms for stable rate adjustments

Competition Scope

For this formal verification competition, the verification focuses on core protocol contracts including:

- **Silo.sol**: Core lending market contract implementing vault logic and state management
- **ShareToken.sol**: ERC20R implementation for debt shares with specialized rebase mechanics
- **Actions**: Protocol action modules handling deposits, withdrawals, borrowing, and liquidations
- **EIP4626 Compliance**: Verification of vault standard compliance for collateral tokens
- **EIP20 Compliance**: Standard token interface compliance for share tokens

---

Table of Contents

Formal Verification Methodology

Certora Formal Verification (FV) provides mathematical proofs of smart contract correctness by verifying code against a formal specification. It complements techniques like testing and fuzzing, which can only sometimes detect bugs based on predefined properties. In contrast, Certora FV examines all possible states and execution paths in a contract.

Simply put, the formal verification process involves crafting properties (similar to writing tests) in CVL language and submitting them alongside compiled Solidity smart contracts to a remote prover. This prover essentially transforms the contract bytecode and rules into a mathematical model and determines the validity of rules.

Types of Properties

When constructing properties in formal verification, we mainly deal with two types: **Invariants** and **Rules**.

Invariants

- Conditions that MUST **always remain true** throughout the contract's lifecycle.
- Process:
    1. Define an initial condition for the contract's state.
    2. Execute an external function.
    3. Confirm the invariant still holds after execution.
- Example: "The total supply MUST always equal the sum of all balances."
- Use Case: Ensures **Valid State** properties - critical state constraints that MUST never be violated.
- Feature: Proven invariants can be reused in other properties with the `requireInvariant` keyword.

Rules

- Flexible checks for specific behaviors or conditions.
- Structure:
    1. Setup: Set assumptions (e.g., "user balance is non-zero").
    2. Execution: Simulate contract behavior by calling external functions.
    3. Verification:
        - Use `assert` to check if a condition is **always true** (e.g., "balance never goes negative").
        - Use `satisfy` to verify a condition is **reachable** (e.g., "a user can withdraw funds").
- Example: "A withdrawal decreases the user's balance."
- Use Case: Verifies a broad range of properties, from simple state changes to complex business logic.

Verification Process

The process is divided into two stages: **Setup** and **Crafting Properties**.

Setup

This stage prepares the contract and prover for verification:

- Resolve external contract calls.
- Simplify complex operations (e.g., math or bitwise calculations) for prover compatibility.
- Install storage hooks to monitor state changes.
- Address prover limitations (e.g., timeouts or incompatibilities).
- Prove **Valid State properties** (invariants) as a foundation for further checks.

Crafting Properties

This stage defines and implements the properties:

- Write properties in **plain English** for clarity.
- Categorize properties by purpose (e.g., Valid State, Variable Transition).
- Use proven invariants as assumptions in **Rules** for efficiency.

Assumptions

Assumptions simplify the verification process and are classified as **Safe** or **Unsafe**. Safe assumptions are backed by valid state invariants or required by the environment. Unsafe made to reduce complexity, potentially limiting coverage.

Safe Assumptions

Protocol Parameter Boundaries

- Fee ranges with specific caps: DAO fee (5-50%), deployer fee (0-15%), liquidation fee (0-30%), flashloan fee (0-15%)
- LTV constraints: Max LTV ≤ LT, LT + liquidation fee ≤ 100%, liquidation target LTV ≤ LT
- Borrower collateral silo must be validly set (0, Silo0, or Silo1)
- Cross-reentrancy guard state is either ENTERED or NOT_ENTERED

ERC20 Accounts

- Token decimals limited to either 0 or between 6-18
- Only 10 different ERC20 accounts modeled

Mathematical Properties

- Monotonicity of compound interest rates over time
- Well-defined rounding behavior in math operations

Testing Boundary Conditions

- Valid timestamp assumptions: `e.block.timestamp > max_uint16 && e.block.timestamp < max_uint48`

Unsafe Assumptions

Value Range Restrictions

- Maximum ERC20 balances limited to `max_uint64` rather than full `uint256` range
- Maximum token supply range limited between `max_uint32` and `max_uint64`
- ERC20 allowances capped at `max_uint128`
- Total supply values capped at `max_uint128`
- Accumulated fee amounts limited to below `max_uint64` to avoid overflow

Token Property Simplifications

- Zero assets correspond to zero shares and vice versa
- Non-zero shares correspond to non-zero tracked assets

Protocol Component Simplifications

- Oracles are disabled
- Solvency check is summarized as `NONDET`

Verification Properties

The verification properties are categorized into the following types:

1. **Valid State (VS)**: System state invariants that MUST always hold
2. **Share Tokens (SHT)**: Common rules for Protected, Collateral and Debt share tokens
3. **Silo (SI)**: Rules governing main Silo contract operations
4. **EIP4626 Compliance (EIP4626)**: Rules ensuring compliance with the EIP-4626 tokenized vault standard
5. **EIP20 Compliance (EIP20)**: Rules ensuring compliance with the EIP-20 token standard
6. **Sanity (SA)**: Check that all external functions stay reachable after valid state assumed

Most of properties have several runs for each property and even for specific methods due to code complexity. Only one run marked in `Links` sections for simplicity.

`EIP4626 Compliance` properties have two unique executable paths: `Collateral` and `Protected` in harnesses.

Each job status linked to a corresponding run in the dashboard with a specific status:

- ✅ completed successfully
- ⚠️ reached global timeout
- ❌ violated

Valid State

The states define the possible values that the system's variables can take. Usually, there can be only one valid state at any given time. Thus, we also check that a system MUST always be in exactly one of its valid states.

Source	Invariant	Description	Links
VS-01	inv_eip20_totalSupplySolvency	Ensures the total supply of tokens equals the sum of all balances	✅
VS-02	inv_crossReentrancyGuardOpenedOnExit	The cross reentrancy guard must remain opened on exit	✅
VS-03	inv_transferWithChecksAlwaysEnabled	The silo's transferWithChecks feature must always remain enabled	✅
VS-04	inv_interestRateTimestampNotInFuture	The interest rate timestamp must never be set in the future	✅
VS-05	inv_borrowerCannotHaveTwoDebts	A borrower must never hold debt in more than one silo at the same time	✅
VS-06	inv_borrowerCannotHaveDebtWithoutCollateralSet	Borrower cannot have debt without collateral set in the config	✅
VS-07	inv_borrowerCannotHaveDebtWithoutCollateralShares	Borrower cannot have debt without collateral shares	✅
VS-08	inv_liquiditySolvency0/1	The Silo's liquidity must cover its protected collateral, collateral, and fees minus any outstanding debt	✅
VS-09	inv_siloMustNotHaveUserAllowances0/1	The Silo contract must never have an allowance to withdraw assets	✅
VS-10	inv_protectedCollateralAlwaysLiquid0/1	Protected collateral must remain fully available for withdrawal	✅
VS-11	inv_zeroCollateralMeansZeroDebt0/1	If the Silo's total collateral is zero, then its total debt must also be zero	✅

Share Tokens

The following rules apply to all share tokens (Protected, Collateral, and Debt) in the Silo protocol.

Source	Rule	Description	Links
SHT-01	share_functionExecutesHooksBasedOnConfig	Functions execute hooks based on their configuration	✅
SHT-02	share_noHookFunctionMustNotExecuteHook	Functions marked as NO_HOOKS_FUNCTIONS must not call hooks	✅
SHT-03	share_hooksShouldBeSynchronized	Share token hook configurations must be synchronized	✅
SHT-04	share_crossReentrancyProtectionNoDoubleCall	No double calls to cross reentrancy protection	✅
SHT-05	share_noStateChangingCallInsideReentrancyEntered	No state-changing calls may occur while already in ENTERED reentrancy state	✅

Source	Rule	Description	Links
SHT-06	share_noMovingSharesInsideReentrancyEntered	Moving shares is not allowed inside a reentrant call	✅
SHT-07	share_allowedReenterFunctionDoNotCallCrossReentrancyGuard	Allowed reentrancy functions never call to CrossReentrancyGuard	✅
SHT-08	share_groupShareChangeRequireGroupTimestamp	Any change in share balances or total supply must have interest up-to-date	✅
SHT-09	share_InterestTimestampAlwaysGrow	Block timestamp never goes backwards	✅
SHT-10	share_enforceHookBeforeAfterOrdering	No storage writes happen before `hookBefore` or after `hookAfter`	✅
SHT-11	share_hooksMustExecuteIfStorageChanged	If storage was changed, then hooks must be called	✅

Silo

The following rules govern the main Silo contract operations in the protocol.

Source	Rule	Description	Links
SI-01	silo_accrueInterestNoSharesChanged	Accruing interest does not affect share balances or supplies	✅
SI-02	silo_possibilityOfCollateralsInTwoSilos	Users can have collateral deposits in both Silo0 and Silo1 simultaneously	✅
SI-03	silo_possibilityOfProtectedInTwoSilos	Users can have protected deposits in both Silo0 and Silo1 simultaneously	✅
SI-04	silo_collateralFunctionsNoAccessOtherVaults	Collateral harness functions must not touch protected/debt storage	✅
SI-05	silo_protectedFunctionsNoAccessOtherVaults	Protected harness functions must not touch debt storage	✅
SI-06	silo_collateralFunctionsAccessOwnStorage	Collateral vault functions can read/write their own storage	✅
SI-07	silo_deployerFeeReceiverCannotBeBlocked	Even if a third party calls the silo, deployer fee receiver can still withdraw	❌

Note: `silo_deployerFeeReceiverCannotBeBlocked` violated due to real issue.

EIP4626 Compliance

These rules ensure the protocol's vaults comply with the EIP-4626 tokenized vault standard.

Source	Rule	Description	Links
EIP4626-01	eip4626_assetIntegrity	Asset function returns the correct underlying token	✅
EIP4626-02	eip4626_assetMustNotRevert	Asset function must not revert	✅
EIP4626-03	eip4626_totalAssetsIntegrity	TotalAssets includes any compounding from yield	✅
EIP4626-04	eip4626_totalAssetsMustNotRevert	TotalAssets function must not revert	✅
EIP4626-05	eip4626_convertToSharesNotIncludeFeesInDeposit	ConvertToShares must not include deposit fees	✅
EIP4626-06	eip4626_convertToSharesNotIncludeFeesInWithdraw	ConvertToShares must not include withdrawal fees	✅
EIP4626-07	eip4626_convertToSharesMustNotDependOnCaller	ConvertToShares must not vary based on caller	✅
EIP4626-08	eip4626_convertToSharesMustNotRevert	ConvertToShares must not revert for reasonable inputs	✅
EIP4626-09	eip4626_convertToSharesRoundTripDoesNotExceed	ConvertToShares must round down towards 0	✅
EIP4626-10	eip4626_convertToSharesNoSlippage	ConvertToShares must not reflect on-chain slippage	✅
EIP4626-11	eip4626_convertToAssetsNotIncludeFeesRedeem	ConvertToAssets must not include redemption fees	✅
EIP4626-12	eip4626_convertToAssetsNotIncludeFeesMint	ConvertToAssets must not include mint fees	✅
EIP4626-13	eip4626_convertToAssetsMustNotDependOnCaller	ConvertToAssets must not vary based on caller	✅
EIP4626-14	eip4626_convertToAssetsMustNotRevert	ConvertToAssets must not revert for reasonable inputs	✅
EIP4626-15	eip4626_convertToAssetsRoundTripDoesNotExceed	ConvertToAssets must round down towards 0	✅
EIP4626-16	eip4626_convertToAssetsNoSlippage	ConvertToAssets must not reflect on-chain slippage	✅
EIP4626-17	eip4626_maxDepositNoHigherThanActual	MaxDeposit must not exceed actual deposit limit	✅
EIP4626-18	eip4626_maxDepositDoesNotDependOnUserBalance	MaxDeposit must not depend on user's asset balance	✅
EIP4626-19	eip4626_maxDepositUnlimitedReturnsMax	MaxDeposit returns max value if no limit exists	✅
EIP4626-20	eip4626_maxDepositMustNotRevert	MaxDeposit must not revert	✅

Source	Rule	Description	Links
EIP4626-21	eip4626_previewDepositNoMoreThanActualShares	PreviewDeposit must not return more shares than actual deposit	✅
EIP4626-22	eip4626_previewDepositMustIgnoreLimits	PreviewDeposit must ignore deposit limits	✅
EIP4626-23	eip4626_previewDepositMustIncludeFees	PreviewDeposit must include deposit fees	✅
EIP4626-24	eip4626_previewDepositMustNotDependOnCaller	PreviewDeposit must not vary based on caller	✅
EIP4626-25	eip4626_previewDepositMayRevertOnlyWithDepositRevert	PreviewDeposit may revert only when deposit would revert	✅
EIP4626-26	eip4626_depositIntegrity	Deposit correctly updates balances and mints shares	✅
EIP4626-27	eip4626_depositToSelfIntegrity	Deposit to self works correctly	✅
EIP4626-28	eip4626_depositRespectsApproveTransfer	Deposit respects EIP-20 approve/transferFrom mechanism	✅
EIP4626-29	eip4626_depositMustRevertIfCannotDeposit	Deposit must revert if full deposit can't be processed	✅
EIP4626-30	eip4626_depositPossibility	Deposit functionality is possible	✅

Note: In this table, `eip4626_` represents both collateral and protected vault implementations of the same rule (e.g., `eip4626_collateral_assetIntegrity` and `eip4626_protected_assetIntegrity`). Links are provided for collateral execution path.

EIP20 Compliance

These rules ensure the protocol's tokens comply with the EIP-20 token standard.

Source	Rule	Description	Links
EIP20-01	eip20_totalSupplyIntegrity	Total token supply value is accurate and consistent	✅
EIP20-02	eip20_balanceOfIntegrity	Account balance queries return correct values	✅
EIP20-03	eip20_allowanceIntegrity	Allowance queries return correct values	✅
EIP20-04	eip20_transferIntegrity	Transfer operations correctly update balances	✅
EIP20-05	eip20_transferMustRevert	Transfer must revert when requirements aren't met	✅

Source	Rule	Description	Links
EIP20-06	eip20_transferSupportZeroAmount	Transfer of 0 value is treated as a normal transfer	❌
EIP20-07	eip20_transferFromIntegrity	TransferFrom correctly updates balances and allowances	✅
EIP20-08	eip20_transferFromMustRevert	TransferFrom must revert when requirements aren't met	✅
EIP20-09	eip20_transferFromSupportZeroAmount	TransferFrom of 0 value is treated as a normal transfer	❌
EIP20-10	eip20_approveIntegrity	Approve correctly sets allowances	✅
EIP20-11	eip20_approveMustRevert	Approve must revert for zero addresses	✅

Note: `eip20_transferSupportZeroAmount` and `eip20_transferFromSupportZeroAmount` are violated due `ZeroTransfer` check in `ShareToken.sol:_update()`:

```
/// @inheritdoc ERC20Upgradeable
function _update(address from, address to, uint256 value) internal virtual override {
    require(value != 0, ZeroTransfer());

    _beforeTokenTransfer(from, to, value);

    ERC20Upgradeable._update(from, to, value);

    _afterTokenTransfer(from, to, value);
}
```

Manual Mutations Testing

This section documents the manual mutations from the Certora FV contest applied to `Silo.sol`, `Actions.sol` and `PartialLiquidation.sol`. Each caught mutation is tested against specific rules to verify that it correctly **violates** under altered conditions.

Silo

Silo_0.sol - inv_protectedCollateralAlwaysLiquid1

This property caught both `Silo_0.sol` and `Actions_5.sol` mutations.

Property: Protected collateral must remain fully available for withdrawal

```
definition protectedCollateralAlwaysLiquid(bool zero) returns bool =
    ghostERC20Balances[ghostTokenX(zero)][ghostSiloX(zero)]
        >= ghostTotalAssets[ghostSiloX(zero)][ASSET_TYPE_PROTECTED()];

invariant inv_protectedCollateralAlwaysLiquid1(env e) protectedCollateralAlwaysLiquid(false)
filtered { f -> !EXCLUDED_OR_VIEW_SILO_FUNCTION(f) }
{ preserved with (env eInv) { requireSameEnv(e, eInv); setupSilo(e); } }
```

Execution: `certoraRun certora/confs/invariants/silo/Silo1_inv_protectedCollateralAlwaysLiquid1_all_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/b8a20e1cf3ad45f6ae419b55cae32ae8?anonymousKey=7745c283f9c0005c5c79b67f4de7bea1cb860ec8

Mutation: mutations/Silo/Silo_0.sol

```
    // mutation: add direct transfer function
    function directTransfer(address _receiver,
        address _token,
        uint256 _amount) external {
        IERC20(_token).safeTransfer(address(_receiver), _amount);
    }
```

After: ❌ https://prover.certora.com/output/52567/fc9f8d78037e4f2b9767322c4d3a4049?anonymousKey=981c51fbe5c7b3be16fdc6a1dd6e6cea58d93f6e

Silo_1.sol - inv_liquiditySolvency1

Property: The Silo's liquidity must cover its protected collateral, collateral, and fees minus any outstanding debt

```
definition liquiditySolvency(bool zero) returns bool =
    ghostERC20Balances[ghostTokenX(zero)][ghostSiloX(zero)] >=
        ghostTotalAssets[ghostSiloX(zero)][ASSET_TYPE_PROTECTED()]
        + ghostTotalAssets[ghostSiloX(zero)][ASSET_TYPE_COLLATERAL()]
        + ghostDaoAndDeployerRevenue[ghostSiloX(zero)]
        - ghostTotalAssets[ghostSiloX(zero)][ASSET_TYPE_DEBT()];

invariant inv_liquiditySolvency1(env e) liquiditySolvency(false)
filtered { f -> !EXCLUDED_OR_VIEW_SILO_FUNCTION(f) }
{ preserved with (env eInv) { requireSameEnv(e, eInv); setupSilo(e); } }
```

Execution: `certoraRun certora/confs/invariants/silo/Silo1_inv_liquiditySolvency1_withdrawCollateral_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/abce6455bb384323a5bb76452152aefd?anonymousKey=4f77be4bb1fe45e1ea1fbe8de99b65107f4e8bf1

Mutation: mutations/Silo/Silo_1.sol

```
function _withdraw(
    uint256 _assets,
```

```
    uint256 _shares,
    address _receiver,
    address _owner,
    address _spender,
    ISilo.CollateralType _collateralType
)
    internal
    virtual
    returns (uint256 assets, uint256 shares)
{
    // MUTATION: Store original total assets before withdrawal
    ISilo.SiloStorage storage $ = SiloStorageLib.getSiloStorage();
    uint256 totalAssetsBefore = $.totalAssets[ISilo.AssetType(uint256(_collateralType))];

    (assets, shares) = Actions.withdraw(
        WithdrawArgs({
            assets: _assets,
            shares: _shares,
            receiver: _receiver,
            owner: _owner,
            spender: _spender,
            collateralType: _collateralType
        })
    );

    // MUTATION: Restore the previous total assets value under certain conditions
    if (_owner == msg.sender && _assets > 1000) {
        $.totalAssets[ISilo.AssetType(uint256(_collateralType))] = totalAssetsBefore;
    }

    if (_collateralType == CollateralType.Collateral) {
        emit Withdraw(msg.sender, _receiver, _owner, assets, shares);
    } else {
        emit WithdrawProtected(msg.sender, _receiver, _owner, assets, shares);
    }
}
```

After: ❌ https://prover.certora.com/output/52567/4dfdd41dacbc4dd4ae348f1b0db3d825?anonymousKey=ea4a1e5b1c0034428e8b28670012730f981d0178

Silo_3.sol - eip4626_collateral_convertToSharesNotIncludeFeesInDeposit

Property: `convertToShares()` MUST NOT be inclusive of any fees that are charged against assets in the Vault (check deposit)

```
rule eip4626_collateral_convertToSharesNotIncludeFeesInDeposit(env e, uint256 assets) {

    // SAFE: Assume valid Silo state
    setupSilo(e);

    // Solve complexity, avoiding unreasonably large input
    require(assets < max_uint64);

    // Another way: previewDeposit() factors in deposit fees, so it will return fewer shares
    //  if a fee is charged

    assert(previewDepositCollateral(e, assets) <= convertToSharesCollateral(e, assets));
}
```

Execution: `certoraRun certora/confs/eip4626_collateral/Silo1_eip4626_collateral_convertToSharesNotIncludeFeesInDeposit_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/06b532d5c41e4a15aba9685f10ef2d35?anonymousKey=88d23c4f46612cdd6376b54f06ceecc528c14c64

Mutation: mutations/Silo/Silo_3.sol

```
function _convertToShares(uint256 _assets, AssetType _assetType) internal view virtual
returns (uint256 shares) {
    (
        uint256 totalSiloAssets, uint256 totalShares
    ) = SiloStdLib.getTotalAssetsAndTotalSharesWithInterest(ShareTokenLib.getConfig(),
_assetType);

    // mutation: incorrectly calculate the number of shares users receive when depositing
assets.
    if (_assetType == AssetType.Collateral) {
        shares = SiloMathLib.convertToShares(
            _assets * 5 / 10, // Reduce the assets by 50% before conversion
            totalSiloAssets,
            totalShares,
            Rounding.DEPOSIT_TO_SHARES,
            _assetType
        );
    } else {
        shares = SiloMathLib.convertToShares(
            _assets,
            totalSiloAssets,
            totalShares,
            _assetType == AssetType.Debt ? Rounding.BORROW_TO_SHARES :
Rounding.DEPOSIT_TO_SHARES,
            _assetType
        );
    }
}
```

After: ❌ https://prover.certora.com/output/52567/064d68466635451bbc9d21bbcf682838/?anonymousKey=3e974ca1524cb83ef9658e5d64fd7d5382b39ac3

Silo_5.sol - share_functionExecutesHooksBasedOnConfig

Property: Check valid action ids inside hooks

```
rule share_functionExecutesHooksBasedOnConfig(env e, method f, calldataarg args)
    filtered { f-> !EXCLUDED_OR_VIEW_SILO_FUNCTION(f) } {

    setupSilo(e);

    require(ghostHookActionAllowAll == true);
    require(ghostBeforeActionId == 0 && ghostAfterActionId == 0);

    f(e, args);

    // Correct id inside match function and hook call
    assert(!NO_HOOKS_FUNCTIONS(f)
        // UNSAFE: TODO - add a support of transfer functions in `ghostSelectorHooks[]`
        && !TRANSFER_ALL_FUNCTIONS(f)
        => (
        ghostExpectedHook == ghostSelectorHooks[to_bytes4(f.selector)]
        && ghostBeforeActionId == ghostExpectedHook
        && ghostAfterActionId == ghostBeforeActionId
    ));
}
```

Execution: `certoraRun certora/confs/share_tokens/silo/Silo1_share_functionExecutesHooksBasedOnConfig_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/512e5c311cbc45c5b4ee445c8a25ba8c?anonymousKey=1172fa21541dd66d74862fbe5f6e442a739830f7

Mutation: mutations/Silo/Silo_5.sol

```
/// @inheritdoc ISilo
function borrowShares(uint256 _shares, address _receiver, address _borrower)
    external
    virtual
    returns (uint256 assets)
{
    uint256 shares;

    // mutation: removed interest accrual before borrowing and set assets to 0
    //(assets, shares) = Actions.borrow(
    //    BorrowArgs({
    //        assets: 0,
    //        shares: _shares,
    //        receiver: _receiver,
    //        borrower: _borrower
    //    })
    //);
```

```
    (assets, shares) = (0, _shares);

    emit Borrow(msg.sender, _receiver, _borrower, assets, shares);
}
```

After: ❌ https://prover.certora.com/output/52567/ad25462773e543d1a381a211cc1bbe22?anonymousKey=95a9174513b9c722b3c524ec60fe3ca397de9942

Silo_6.sol - eip4626_collateral_depositIntegrity

Property: `deposit()` mints shares Vault shares to receiver by depositing exactly assets of underlying tokens

```
rule eip4626_collateral_depositIntegrity(env e, uint256 assets, address receiver) {

    // SAFE: Assume valid Silo state
    setupSilo(e);

    // Pre-state checks
    mathint vaultAssetsPrev    = ghostERC20Balances[ghostToken1][currentContract];
    mathint callerBalancePrev  = ghostERC20Balances[ghostToken1][ghostCaller];
    mathint receiverSharesPrev = ghostERC20Balances[currentContract][receiver];
    mathint vaultSharesSupplyPrev = ghostERC20TotalSupply[currentContract];

    // Attempt deposit
    mathint shares = depositCollateral(e, assets, receiver);

    // Post-state checks

    // The vault's asset balance must have increased by exactly `assets`
    mathint vaultAssetsPost = ghostERC20Balances[ghostToken1][currentContract];
    assert(vaultAssetsPost == vaultAssetsPrev + assets);

    // The caller's asset balance must have decreased by exactly `assets`
    mathint callerBalancePost = ghostERC20Balances[ghostToken1][ghostCaller];
    assert(callerBalancePost == callerBalancePrev - assets);

    // The receiver's share balance must have increased by `shares`
    mathint receiverSharesPost = ghostERC20Balances[currentContract][receiver];
    assert(receiverSharesPost == receiverSharesPrev + shares);

    // The vault's total supply of shares must have increased by `shares`
    mathint vaultSharesSupplyPost = ghostERC20TotalSupply[currentContract];
    assert(vaultSharesSupplyPost == vaultSharesSupplyPrev + shares);
}
```

Execution: `certoraRun certora/confs/eip4626_collateral/Silo1_eip4626_collateral_depositIntegrity_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/1fd3575d788c409582eff7fac4ab6fdd?anonymousKey=79ff50c38bc836d3343470dd07c9108948ddee09

Mutation: mutations/Silo/Silo_6.sol

```
function _deposit(
    uint256 _assets,
    uint256 _shares,
    address _receiver,
    ISilo.CollateralType _collateralType
)
    internal
    virtual
    returns (uint256 assets, uint256 shares)
{
    (
        assets, shares
    ) = Actions.deposit(_assets, _shares, _receiver, _collateralType);

    // mutation: set assets and shares to the original values
    assets = _assets;
    shares = _shares;

    if (_collateralType == CollateralType.Collateral) {
        emit Deposit(msg.sender, _receiver, assets, shares);
    } else {
        emit DepositProtected(msg.sender, _receiver, assets, shares);
    }
}
```

After: ❌ https://prover.certora.com/output/52567/33e60ced6a0a4311b32b58ee3f56a45d?anonymousKey=ebcbf5071926c317a68c29d8c9173a5bfa80f3b7

Silo_8.sol - eip4626_collateral_previewMintMustIncludeFees

Property: `previewMint()` MUST be inclusive of deposit fees. Integrators should be aware of the existence of deposit fees

```
rule eip4626_collateral_previewMintMustIncludeFees(env e, uint256 shares) {

    // SAFE: Assume valid Silo state
    setupSilo(e);

    // Solve complexity, avoiding unreasonably large input
    require(shares < max_uint64);

    mathint pm = previewMintCollateral(e, shares);
    mathint cta = convertToAssetsCollateral(e, shares);

    // Because deposit fees => user needs more assets => pm >= cta
    // If no fees, pm == cta. But never < cta.
    assert(pm >= cta);
}
```

Execution: `certoraRun certora/confs/eip4626_collateral/Silo1_eip4626_collateral_previewMintMustIncludeFees_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/d2180d2d44094514ae08859f99906cb6?anonymousKey=bb0df443ae25502be21d288a62c191c889e8cae5

Mutation: mutations/Silo/Silo_8.sol

```
function _convertToAssets(uint256 _shares, AssetType _assetType) internal view virtual
returns (uint256 assets) {
    // mutation: removed the calculation of total assets and shares and set assets to the
original shares
    // (
    //     uint256 totalSiloAssets, uint256 totalShares
    // ) = SiloStdLib.getTotalAssetsAndTotalSharesWithInterest(ShareTokenLib.getConfig(),
_assetType);

    // assets = SiloMathLib.convertToAssets(
    //     _shares,
    //     totalSiloAssets,
    //     totalShares,
    //     _assetType == AssetType.Debt ? Rounding.BORROW_TO_ASSETS :
Rounding.DEPOSIT_TO_ASSETS,
    //     _assetType
    // );

    assets = _shares;
}

function _convertToShares(uint256 _assets, AssetType _assetType) internal view virtual
returns (uint256 shares) {
    // mutation: removed the calculation of total assets and shares and set shares to the
original assets
    // (
    //     uint256 totalSiloAssets, uint256 totalShares
    // ) = SiloStdLib.getTotalAssetsAndTotalSharesWithInterest(ShareTokenLib.getConfig(),
_assetType);

    // shares = SiloMathLib.convertToShares(
    //     _assets,
    //     totalSiloAssets,
    //     totalShares,
    //     _assetType == AssetType.Debt ? Rounding.BORROW_TO_SHARES :
Rounding.DEPOSIT_TO_SHARES,
    //     _assetType
    // );

    shares = _assets;
}
```

After: ❌ https://prover.certora.com/output/52567/1766bc9b9d5940408f45dd80425f49ef/?anonymousKey=426e861148c73604810fe5e0a1d48f04dea2d6c7

Actions

Actions_0.sol - inv_crossReentrancyGuardOpenedOnExit

Property: The cross reentrancy guard must remain opened on exit

```
invariant inv_crossReentrancyGuardOpenedOnExit(env e)
    ghostCrossReentrantStatus == NOT_ENTERED()
filtered {
    // SAFE: Ignore turning on protection function
    f -> f.selector != 0x9dd41330   // Config.turnOnReentrancyProtection()
    && !EXCLUDED_OR_VIEW_SILO_FUNCTION(f)
    }
{ preserved with (env eInv) { requireSameEnv(e, eInv); setupSilo(e); } }
```

Execution: `certoraRun certora/confs/invariants/silo/Silo1_inv_crossReentrancyGuardOpenedOnExit_switchCollateralToThisSilo_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/1284a400890e47c9b0969d877a817103?anonymousKey=c605fd9323e727e874c923371a01adaf7c75e604

Mutation: mutations/Actions/Actions_0.sol

```
function switchCollateralToThisSilo() external {
    IShareToken.ShareTokenStorage storage _shareStorage =
ShareTokenLib.getShareTokenStorage();

    uint256 action = Hook.SWITCH_COLLATERAL;

    if (_shareStorage.hookSetup.hooksBefore.matchAction(action)) {
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(
            address(this), action, abi.encodePacked(msg.sender)
        );
    }

    ISiloConfig siloConfig = _shareStorage.siloConfig;

    require(siloConfig.borrowerCollateralSilo(msg.sender) != address(this),
ISilo.CollateralSiloAlreadySet());

    siloConfig.turnOnReentrancyProtection();
    siloConfig.setThisSiloAsCollateralSilo(msg.sender);

    ISiloConfig.ConfigData memory collateralConfig;
    ISiloConfig.ConfigData memory debtConfig;

    (collateralConfig, debtConfig) = siloConfig.getConfigsForSolvency(msg.sender);

    if (debtConfig.silo != address(0)) {
        siloConfig.accrueInterestForBothSilos();
        _checkSolvencyWithoutAccruingInterest(collateralConfig, debtConfig, msg.sender);
```

```
    // mutation: only turn off reentrancy protection when debtConfig.silo is not address(0)
    } else {
        siloConfig.turnOffReentrancyProtection();
    }


    if (_shareStorage.hookSetup.hooksAfter.matchAction(action)) {
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).afterAction(
            address(this), action, abi.encodePacked(msg.sender)
        );
    }
}
```

After: ❌ https://prover.certora.com/output/52567/0b92c44fc62345de8eda4f8a1e3f7735?anonymousKey=055f0ac35fdfec46e7c7623cc1670a3486d89676

Actions_4.sol - share_groupShareChangeRequireGroupTimestamp

Property: Any change in share balances or total supply must have interest up-to-date (same block)

```
rule share_groupShareChangeRequireGroupTimestamp(env e, method f, calldataarg args, address user)
    filtered {
    // SAFE: Can be executed by Silo only
    f -> f.selector != 0xc6c3bbe6   // ShareDebtToken.mint()
    && f.selector != 0xf6b911bc     // ShareDebtToken.burn()
    // SAFE: Can be executed by HookReceiver only
    && f.selector != 0xd985616c     // ShareDebtToken.forwardTransferFromNoChecks()
    && !EXCLUDED_OR_VIEW_SILO_FUNCTION(f)
    // UNSOUND: we can transfer collateral shares which are not used as a collateral
    && f.selector != 0xa9059cbb // transfer()
    && f.selector != 0x23b872dd // transferFrom()
    } {

    setupSilo(e);

    // --- Group0: (Debt0, Collateral0, Protected0) ---

    // Record Group0 share balances for `user` before
    mathint debt0BalBefore = ghostERC20Balances[_Debt0][user];
    mathint coll0BalBefore = ghostERC20Balances[_Collateral0][user];
    mathint prot0BalBefore = ghostERC20Balances[_Protected0][user];

    // Record total supply for Group0 share tokens before
    mathint debt0SupplyBefore = ghostERC20TotalSupply[_Debt0];
    mathint coll0SupplyBefore = ghostERC20TotalSupply[_Collateral0];
    mathint prot0SupplyBefore = ghostERC20TotalSupply[_Protected0];

    // --- Group1: (Debt1, Collateral1, Protected1) ---

    // Record Group1 share balances for `user` before
    mathint debt1BalBefore = ghostERC20Balances[_Debt1][user];
    mathint coll1BalBefore = ghostERC20Balances[_Collateral1][user];
```

```
    mathint prot1BalBefore = ghostERC20Balances[_Protected1][user];

    // Record total supply for Group1 share tokens before
    mathint debt1SupplyBefore = ghostERC20TotalSupply[_Debt1];
    mathint coll1SupplyBefore = ghostERC20TotalSupply[_Collateral1];
    mathint prot1SupplyBefore = ghostERC20TotalSupply[_Protected1];

    f(e, args);

    // After
    mathint debt0BalAfter = ghostERC20Balances[_Debt0][user];
    mathint coll0BalAfter = ghostERC20Balances[_Collateral0][user];
    mathint prot0BalAfter = ghostERC20Balances[_Protected0][user];
    mathint debt0SupplyAfter = ghostERC20TotalSupply[_Debt0];
    mathint coll0SupplyAfter = ghostERC20TotalSupply[_Collateral0];
    mathint prot0SupplyAfter = ghostERC20TotalSupply[_Protected0];
    mathint silo0InterestAfter = ghostInterestRateTimestamp[_Silo0];

    mathint debt1BalAfter = ghostERC20Balances[_Debt1][user];
    mathint coll1BalAfter = ghostERC20Balances[_Collateral1][user];
    mathint prot1BalAfter = ghostERC20Balances[_Protected1][user];
    mathint debt1SupplyAfter = ghostERC20TotalSupply[_Debt1];
    mathint coll1SupplyAfter = ghostERC20TotalSupply[_Collateral1];
    mathint prot1SupplyAfter = ghostERC20TotalSupply[_Protected1];
    mathint silo1InterestAfter = ghostInterestRateTimestamp[_Silo1];

    bool changedGroup0 = (
        debt0BalBefore != debt0BalAfter
     || coll0BalBefore != coll0BalAfter
     || prot0BalBefore != prot0BalAfter
     || debt0SupplyBefore != debt0SupplyAfter
     || coll0SupplyBefore != coll0SupplyAfter
     || prot0SupplyBefore != prot0SupplyAfter
    );

    bool changedGroup1 = (
        debt1BalBefore != debt1BalAfter
     || coll1BalBefore != coll1BalAfter
     || prot1BalBefore != prot1BalAfter
     || debt1SupplyBefore != debt1SupplyAfter
     || coll1SupplyBefore != coll1SupplyAfter
     || prot1SupplyBefore != prot1SupplyAfter
    );

    // If shares for groups changed, silo’s interest must have updated
    assert(changedGroup0 => silo0InterestAfter == e.block.timestamp);
    assert(changedGroup1 => silo1InterestAfter == e.block.timestamp);
}
```

Execution: `certoraRun`

`certora/confs/share_tokens/silo/Silo1_share_groupShareChangeRequireGroupTimestamp_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/418feeadc4fb4176bd2eb864a52b81b8?anonymousKey=5a521a3ce312f946a0377226df016c52aff4f6e4

Mutation: mutations/Actions/Actions_4.sol

```
function repay(
    uint256 _assets,
    uint256 _shares,
    address _borrower,
    address _repayer
)
    external
    returns (uint256 assets, uint256 shares)
{
    IShareToken.ShareTokenStorage storage _shareStorage =
ShareTokenLib.getShareTokenStorage();

    if (_shareStorage.hookSetup.hooksBefore.matchAction(Hook.REPAY)) {
        bytes memory data = abi.encodePacked(_assets, _shares, _borrower, _repayer);
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(address(this),
Hook.REPAY, data);
    }

    ISiloConfig siloConfig = _shareStorage.siloConfig;

    siloConfig.turnOnReentrancyProtection();
    // MUTATION: Skip interest accrual to allow repaying without accounting for accrued
interest
    // siloConfig.accrueInterestForSilo(address(this));

    (address debtShareToken, address debtAsset) =
siloConfig.getDebtShareTokenAndAsset(address(this));

    (assets, shares) = SiloLendingLib.repay(
        IShareToken(debtShareToken), debtAsset, _assets, _shares, _borrower, _repayer
    );

    siloConfig.turnOffReentrancyProtection();

    if (_shareStorage.hookSetup.hooksAfter.matchAction(Hook.REPAY)) {
        bytes memory data = abi.encodePacked(_assets, _shares, _borrower, _repayer, assets,
shares);
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).afterAction(address(this),
Hook.REPAY, data);
    }
}
```

After: ❌ https://prover.certora.com/output/52567/b944a0cd5ba249ea95173e9d09fd0ec3?anonymousKey=cfa890518e32f230c4382c3542a4f2e6a1ae012f

Actions_5.sol - inv_protectedCollateralAlwaysLiquid1

This property caught both `Silo_0.sol` and `Actions_5.sol` mutations.

Property: Protected collateral must remain fully available for withdrawal

```
definition protectedCollateralAlwaysLiquid(bool zero) returns bool =
    ghostERC20Balances[ghostTokenX(zero)][ghostSiloX(zero)]
        >= ghostTotalAssets[ghostSiloX(zero)][ASSET_TYPE_PROTECTED()];

invariant inv_protectedCollateralAlwaysLiquid1(env e) protectedCollateralAlwaysLiquid(false)
filtered { f -> !EXCLUDED_OR_VIEW_SILO_FUNCTION(f) }
{ preserved with (env eInv) { requireSameEnv(e, eInv); setupSilo(e); } }
```

Execution: `certoraRun certora/confs/invariants/silo/Silo1_inv_protectedCollateralAlwaysLiquid1_others_verified.conf --method "withdrawFees()"`

Before: ✅ https://prover.certora.com/output/52567/4263bedcf9c84c058dfd62932528d2db?anonymousKey=1d00c0afb0f9a21da62f9572b146c7d9d08c39fc

Mutation: mutations/Actions/Actions_5.sol

```
function withdrawFees(ISilo _silo) external returns (uint256 daoRevenue, uint256
deployerRevenue) {
    ISiloConfig siloConfig = ShareTokenLib.siloConfig();
    siloConfig.turnOnReentrancyProtection();

    ISilo.SiloStorage storage $ = SiloStorageLib.getSiloStorage();

    uint256 earnedFees = $.daoAndDeployerRevenue;
    require(earnedFees != 0, ISilo.EarnedZero());

    (
        address daoFeeReceiver,
        address deployerFeeReceiver,
        uint256 daoFee,
        uint256 deployerFee,
        address asset
    ) = SiloStdLib.getFeesAndFeeReceiversWithAsset(_silo);

    uint256 availableLiquidity;
    uint256 siloBalance = IERC20(asset).balanceOf(address(this));

    uint256 protectedAssets = $.totalAssets[ISilo.AssetType.Protected];

    // MUTATION: Skip available liquidity check to allow withdrawing more fees than
available
    // unchecked { availableLiquidity = protectedAssets > siloBalance ? 0 : siloBalance -
protectedAssets; }
    availableLiquidity = siloBalance;  // Always use full balance

    // require(availableLiquidity != 0, ISilo.NoLiquidity());
```

```
    if (earnedFees > availableLiquidity) earnedFees = availableLiquidity;

    // we will never underflow because earnedFees max value is `daoAndDeployerRevenue`
    unchecked { $.daoAndDeployerRevenue -= uint192(earnedFees); }

    ...
}
```

After: ❌ https://prover.certora.com/output/52567/4457fc18c6854f2f972cc3ae1b1b0601?anonymousKey=ca64f1864275291f43d55054d2a0a034a8d2931a

Actions_P.sol - sanity_others

Property: Reachability of external functions with valid state assumptions

```
rule sanity_others(method f, env e, calldataarg args) filtered { f->
    f.selector != sig:transitionCollateralFromCollateral(uint256,address).selector
    && f.selector != sig:transitionCollateralFromProtected(uint256,address).selector
    && f.selector != sig:redeemCollateral(uint256,address,address).selector
    && f.selector != sig:redeemProtected(uint256,address,address).selector
    && f.selector != sig:withdrawCollateral(uint256,address,address).selector
    && f.selector != sig:withdrawProtected(uint256,address,address).selector
    && f.selector != sig:borrowShares(uint256,address,address).selector
    && f.selector != sig:borrow(uint256,address,address).selector
    && f.selector != sig:borrowSameAsset(uint256,address,address).selector
    && !EXCLUDED_SILO_FUNCTION(f)
} {
    setupSilo(e);
    f(e, args);
    satisfy(true);
}
```

Execution: `certoraRun certora/confs/sanity/silo/Silo1_sanity_others_verified.conf --method "flashLoan(address,address,uint256,bytes)"`

Before: ✅ https://prover.certora.com/output/52567/678ddf4140b54491a03b46ccc1d9bac4?anonymousKey=24d05b65f6385d721d1af4bd7b1af2407059ce82

Mutation: mutations/Actions/Actions_P.sol

```
function flashLoan(
    IERC3156FlashBorrower _receiver,
    address _token,
    uint256 _amount,
    bytes calldata _data
)
    external
    returns (bool success)
{
    require(_amount != 0, ISilo.ZeroAmount());
```

```
    IShareToken.ShareTokenStorage storage _shareStorage =
ShareTokenLib.getShareTokenStorage();

    if (_shareStorage.hookSetup.hooksBefore.matchAction(Hook.FLASH_LOAN)) {
        bytes memory data = abi.encodePacked(_receiver, _token, _amount);
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(address(this),
Hook.FLASH_LOAN, data);
    }

    // flashFee will revert for wrong token
    uint256 fee = SiloStdLib.flashFee(_shareStorage.siloConfig, _token, _amount);

    require(fee <= type(uint192).max, FeeOverflow());
    // this check also verify if token is correct
    require(_amount <= Views.maxFlashLoan(_token), FlashLoanNotPossible());

    // cast safe, because we checked `fee > type(uint192).max`
    SiloStorageLib.getSiloStorage().daoAndDeployerRevenue += uint192(fee);

    // mutation: replace "_receiver" with "this"
    IERC20(_token).safeTransfer(address(this), _amount);

    require(
        _receiver.onFlashLoan(msg.sender, _token, _amount, fee, _data) ==
_FLASHLOAN_CALLBACK,
        ISilo.FlashloanFailed()
    );

    IERC20(_token).safeTransferFrom(address(_receiver), address(this), _amount + fee);

    if (_shareStorage.hookSetup.hooksAfter.matchAction(Hook.FLASH_LOAN)) {
        bytes memory data = abi.encodePacked(_receiver, _token, _amount, fee);
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).afterAction(address(this),
Hook.FLASH_LOAN, data);
    }

    success = true;
}
```

After: ❌ https://prover.certora.com/output/52567/3ae1ca1f1e0b40bc82fae0fc8c9b1bfd?anonymousKey=030b571774b5c7868eaf277b1043eb20e1605a05

PartialLiquidation

PartialLiquidation_1.sol - sanity_liquidationCall_noSToken_noBypass_protectedAllowed

Property: Reachability of external functions with valid state assumptions

```
rule sanity_liquidationCall_noSToken_noBypass_protectedAllowed(
    env e,
    address _borrower,
    uint256 _maxDebtToCover
) {
    setupSilo(e);
    liquidationCall_noSToken_noBypass_protectedAllowed(e, _borrower, _maxDebtToCover);
    satisfy(true);
}
```

Execution: `certoraRun certora/confs/sanity/hook_to_silo/Hook_sanity_liquidationCall_noSToken_noBypass_protectedAllowed_verified.conf`

Before: ✅ https://prover.certora.com/output/52567/9eb5e888bec14eb6a103222270495775/?anonymousKey=fdb9d3618a68e949eed4971ae4a32782d17e80a5

Mutation: mutations/PartialLiquidation/PartialLiquidation_1.sol

```
/// @inheritdoc IPartialLiquidation
function liquidationCall( // solhint-disable-line function-max-lines, code-complexity
    address _collateralAsset,
    address _debtAsset,
    address _borrower,
    uint256 _maxDebtToCover,
    bool _receiveSToken
)
    external
    virtual
    returns (uint256 withdrawCollateral, uint256 repayDebtAssets)
{
    ISiloConfig siloConfigCached = siloConfig;

    require(address(siloConfigCached) != address(0), EmptySiloConfig());
    require(_maxDebtToCover != 0, NoDebtToCover());

    // mutation: turn off reentrancy protection instead of turning on
    siloConfigCached.turnOffReentrancyProtection();

    (
        ISiloConfig.ConfigData memory collateralConfig,
        ISiloConfig.ConfigData memory debtConfig
    ) = _fetchConfigs(siloConfigCached, _collateralAsset, _debtAsset, _borrower);

    ...
}
```

After: ❌ https://prover.certora.com/output/52567/4a978aca7785463a80fc657cd4829664/?anonymousKey=0f2bf8f873b3e032cb59a7c83a82affb155881d7

Setup and Execution Instructions

Certora Prover Installation

For step-by-step installation steps refer to this setup tutorial.

Verification Execution

Due to the complexity of some verification rules, certain properties must be run individually to avoid timeouts, while others can be executed together. Bash scripts are provided to streamline the process. Run all commands from the root directory of the project.

1. **Valid State (VS)**: System state invariants that MUST always hold

```
./certora/scripts/invariants/run_all.sh
```

2. **Share Tokens (SHT)**: Common rules for Protected, Collateral and Debt share tokens

```
./certora/scripts/share_tokens/run_all.sh
./certora/scripts/share_tokens_split/run_all.sh  # For storage splitting variant
```

3. **Silo (SI)**: Rules governing main Silo contract operations

```
./certora/scripts/silo/run_silo.sh
./certora/scripts/silo_split/run_silo.sh  # For storage splitting variant
```

4. **EIP4626 Compliance (EIP4626)**: Rules ensuring compliance with the EIP-4626 tokenized vault standard

```
./certora/scripts/eip4626_collateral/run_silo.sh  # For collateral token compliance
./certora/scripts/eip4626_protected/run_silo.sh   # For protected token compliance
```

5. **EIP20 Compliance (EIP20)**: Rules ensuring compliance with the EIP-20 token standard

```
./certora/scripts/eip20/run_all.sh
```

6. **Sanity (SA)**: Check that all external functions stay reachable after valid state assumed

```
./certora/scripts/sanity/run_all.sh
```

To regenerate all configurations and run all verifications in sequence:

```
./certora/scripts/gen_all_confs.sh  # Generate all configurations
./certora/scripts/mutate_all.sh     # Run mutation testing
```

Individual component verification is also available for specific contract types:

```
# For Silo contracts
./certora/scripts/silo/run_silo.sh
```

```
# For Protected token
./certora/scripts/share_tokens/run_protected.sh
./certora/scripts/eip20/run_protected.sh
./certora/scripts/sanity/run_protected.sh

# For Debt token
./certora/scripts/share_tokens/run_debt.sh
./certora/scripts/eip20/run_debt.sh
./certora/scripts/sanity/run_debt.sh

# For Hook functionality
./certora/scripts/invariants/run_hook.sh
./certora/scripts/sanity/run_hook.sh
```
