Formal Verification Report: Licredity v1 Core

- Repository: https://github.com/Cyfrin/audit-2025-08-licredity
- Latest Commit Hash: 5effe9b
- Scope: core/
- Date: August 2025
- Author: @alexzoid (@cyfrin private formal verification engagement)
- Certora Prover version: 8.1.1

Table of Contents

About Licredity

Licredity is a permissionless, self-custodial credit protocol built on Uniswap v4 that enables borrowers to mint interest-bearing debt tokens against collateral and deploy them for leveraged positions. The protocol creates a credit market where liquidity providers earn enhanced yields through proactive interest donations directly to Uniswap v4 pools.

Key features include:

- Multi-asset collateral support (ERC20 and ERC721 tokens)
- Dynamic interest rates based on pool price dynamics
- Automated liquidation mechanism for unhealthy positions
- LP yield enhancement through direct interest donations
- Continuous position health monitoring for system stability

Verification Scope

For this formal verification, the focus is on core protocol contract and dependencies:

- **Licredity.sol**: The main contract that provides core functionalities including position management (open/close), collateral deposits/withdrawals (fungible and non-fungible assets), debt issuance and repayment, interest accrual and collection, liquidation mechanisms, and Uniswap v4 hook integration for LP yield enhancement through interest donations

---

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 voting power MUST always equal the sum of all individual NFT voting powers."
- 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 has a veNFT with non-zero voting power").
    2. Execution: Simulate contract behavior by calling external functions.
    3. Verification:
        - Use `assert` to check if a condition is **always true** (e.g., "votes never exceed voting power").
        - Use `satisfy` to verify a condition is **reachable** (e.g., "a user can successfully vote").
- Example: "Voting for a gauge increases the gauge's weight proportionally."
- 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 and dependencies.
- 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).

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).
- Prove valid state invariants as a foundation for further rules

Assumptions

Assumptions are constraints applied during verification to make the problem tractable for the prover. They are classified as **Safe** (no impact on security guarantees) or **Unsafe** (may limit coverage).

Safe Assumptions

These assumptions reflect real-world constraints or simplify non-critical aspects without compromising verification validity:

- Timestamp bounds: Current timestamp limited to realistic values (up to year 2100) to prevent overflow while covering all practical scenarios
- Utility functions removed from verification: `extsload()` functions (external storage reading) and `onERC721Received()` (ERC721 receiver callback) - not part of core protocol logic
- Core functions replaced with harness implementations: Functions requiring complex state management replaced with harness versions for verification
    - 2-step staging flow: `unlock()`, `stageFungible()`, `stageNonFungible()`, `exchangeFungible()`, `depositFungible()`, `depositNonFungible()`
    - Functions simulating unlock flow: `withdrawFungible()`, `withdrawNonFungible()`, `increaseDebtShare()`, `seize()`
- ERC20/ERC721 tokens modeled in CVL: External token contracts abstracted with CVL implementations
    - Token balances capped at `max_uint128` to avoid arithmetic overflows
    - Standard token behaviors (transfer, approve, etc.) modeled accurately

Unsafe Assumptions

These assumptions reduce verification scope to avoid prover timeouts but potentially may miss edge cases:

- Position limits:
    - Single position verification: Only one position is analyzed in detail
    - Multi-position: Limited to 3 positions for bounded verification
- Array size limits:
    - Up to 3 fungibles per position
    - Up to 3 Non-fungibles per position
- Token holder limits:
    - ERC20 tokens supporting up to 4 tokens and 5 users each
    - ERC721 tokens supporting up to 3 users and 5 tokens each

- Uniswap V4 interactions excluded: Pool manager operations (`donate()`, `swap()`, `sync()`, `settle()`, `take()`) summarized as NONDET - excluded from verification scope
- Oracle calls summarized as NONDET: Price quotes (`quotePrice()`, `quoteFungibles()`, `quoteNonFungibles()`) return non-deterministic values - no assumptions about return values

---

Verification Properties

The verification properties are categorized into five distinct types:

1. **Valid State (VS)**: System-wide invariants that MUST always hold true. These properties define the fundamental constraints of the protocol, such as accounting consistency and structural integrity. Once proven, these invariants serve as trusted assumptions in other properties via `requireInvariant`, reducing verification complexity.
2. **State Transition (ST)**: Properties that verify the correctness of transitions between valid states. Building upon the valid state invariants, these properties ensure the protocol's state machine operates correctly and that state changes are both authorized and sequentially valid.

Most properties have multiple verification runs with different configurations to handle code complexity and timeout constraints. Links to specific Certora Prover runs are provided for each property, with status indicators:

- ✅ Verified successfully
- ⚠️ Timeout (property holds but requires optimization)
- ❌ Violated (indicates a potential issue)

Valid State

Valid State properties define the fundamental invariants that must always hold true throughout the protocol's lifecycle. These properties are organized by contract and proven as invariants, meaning they are checked to hold after every possible function execution from any valid initial state.

Single position

Property	Name	Description	Links	Notes
VS-LI-01	lastInterestCollectionNotInFuture	Last interest collection timestamp cannot be in the future	✅	
VS-LI-02	liquidityOnsetsNotInFuture	Liquidity onset timestamps cannot be in the future	✅	
VS-LI-03	positionsWithDataMustHaveOwner	Positions with any data must have a non-zero owner	❌	Issue: Missing zero address check for recipient
VS-LI-04	totalDebtNotExceedLimit	Total debt balance must never exceed the configured debt limit	✅	

Property	Name	Description	Links	Notes
VS-LI-05	debtOutstandingWithinSupply	Debt amount outstanding must not exceed total supply	❌	Issues: EIP-20 zero address, Missing zero address check
VS-LI-06	noDebtTokenAllowances	Licredity must have no debt token allowances	✅	
VS-LI-07	outstandingDebtPairedWithAvailableBase	Outstanding debt and available base must be paired	✅	
VS-LI-08	singlePositionDebtSolvency	Total debt shares must equal initial locked shares plus position debt share	✅	
VS-LI-09	fungibleArrayElementsBeyondLengthAreEmpty	Fungible array elements beyond length must be zero	✅	
VS-LI-10	allFungiblesAreUnique	All fungibles in a position must be unique	✅	
VS-LI-11	fungiblesHaveNonZeroBalance	Fungibles in array must have non-zero balance	❌	Issues: Zero-balance entries, Missing zero delta check
VS-LI-12	fungiblesHaveCorrectIndex	Fungibles in array must have matching index in fungibleStates	❌	Issue: Swap-and-pop without index fix-up
VS-LI-13	fungibleIndexesWithinBounds	Fungible indexes must be within valid bounds	❌	Issue: Swap-and-pop without index fix-up
VS-LI-14	fungibleStatesPointToCorrectPosition	Fungibles with state must be at their claimed position	❌	Issue: Swap-and-pop without index fix-up
VS-LI-15	fungiblePositionBalancesBacked	Fungible balances in positions must be backed by contract balance	✅	
VS-LI-16	baseTokenPositionBalancesBacked	Base tokens reserved for exchange must be backed	✅	
VS-LI-17	nonFungibleArrayElementsBeyondLengthAreEmpty	NonFungible array elements beyond length must be zero	✅	

Property	Name	Description	Links	Notes
VS-LI-18	allNonFungiblesAreUnique	All nonFungibles in a position must be unique	✅	
VS-LI-19	nonFungiblesOwnedByLicredity	Licredity must own all NonFungibles stored in positions	✅	

Multi-position

Property	Name	Description	Links	Notes
VS-LI-20	invalidPositionsAreEmpty	Data for positions outside the bounded set must be empty	✅	
VS-LI-21	positionDebtSolvency	Total debt shares must equal initial shares plus sum of all position debts	✅	
VS-LI-22	positionsBeyondCountAreEmpty	Positions beyond positionCount must be completely empty	✅	

State Transition

State Transition properties verify the correctness of transitions between valid states. These properties ensure that state changes occur only under the right conditions, such as calls to specific functions or time elapsing.

Single position

Specification: state_transition_single.spec

Property	Name	Description	Links	Notes
ST-LI-01	transitionPositionModificationRequiresRegistration	Position modifications that could reduce health must register in locker for validation	✅	
ST-LI-02	transitionDebtChangesRequireInterestAccrual	Debt operations must collect interest when time has elapsed	❌	Issue: Interest accrual bypass

EIP20 Compliance

EIP20 Compliance properties verify that the Licredity contract correctly implements the ERC-20 token standard as defined in EIP-20.

Specification: eip20_compliance.spec

Property	Name	Description	Links	Notes
EIP20-01	eip20_totalSupplyIntegrity	Verify totalSupply() returns correct total token supply	✅	
EIP20-02	eip20_balanceOfIntegrity	Verify balanceOf() returns correct balance for any account	✅	
EIP20-03	eip20_allowanceIntegrity	Verify allowance() returns correct spending allowance	✅	
EIP20-04	eip20_transferIntegrity	Verify transfer() correctly updates balances and maintains invariants	❌	Issue: EIP-20 transfer accepts zero address
EIP20-05	eip20_transferMustRevert	Verify transfer() reverts in invalid conditions	❌	Issue: EIP-20 transfer accepts zero address
EIP20-06	eip20_transferSupportZeroAmount	Verify transfer() handles zero amount transfers correctly	✅	
EIP20-07	eip20_transferFromIntegrity	Verify transferFrom() correctly updates balances, allowances and maintains invariants	❌	Issue: EIP-20 transfer accepts zero address
EIP20-08	eip20_transferFromMustRevert	Verify transferFrom() reverts in invalid conditions	❌	Issue: EIP-20 transfer accepts zero address
EIP20-09	eip20_transferFromSupportZeroAmount	Verify transferFrom() handles zero amount transfers correctly	✅	
EIP20-10	eip20_approveIntegrity	Verify approve() correctly sets allowances without affecting balances	✅	
EIP20-11	eip20_approveMustRevert	Verify approve() reverts in invalid conditions	✅	

Real Issues Properties

This section documents vulnerabilities discovered during the manual audit (including issues by all participants) and formal verification process. Each issue demonstrates how formal properties detected the vulnerability and confirmed its resolution after applying the fix.

[CRITICAL] Swap-and-pop without index fix-up corrupts `Position`'s fungible array (#22)

`Position` tracks fungible collateral in **two places** that must stay in sync:

```
struct Position {
    address owner;
    uint256 debtShare;
    Fungible[] fungibles;                       // compact list of held assets
    NonFungible[] nonFungibles;
    mapping(Fungible => FungibleState) fungibleStates; // per-asset {index,balance}
}
```

Invariant: for every asset `a` in `fungibles[k]`, we must have `fungibleStates[a].index == k+1` (the library uses 1-based indexes; `index == 0` means "not present").

When an asset's balance goes to zero, `Position::removeFungible` tries to "swap-and-pop": move the last array element into the removed slot and shrink the array. The code does the array move, but forgets to update the moved element's cached index in `fungibleStates`.

❌Violated in `fungiblesHaveCorrectIndex`: https://prover.certora.com/output/52567/2585e6785b3e490d9dc65e1ad48adde1/?anonymousKey=c0fcee829b33938b9d64553a37ae6f7d0132c083

```
// VS-LI-12: Fungibles in array must have matching index in fungibleStates
// For every fungible in the fungibles array, the index stored in fungibleStates must be
array position + 1 (1-based)
invariant fungiblesHaveCorrectIndex(env e)
    forall mathint i.
        i >= 0 && i < ghostLiPositionFungiblesLength
            => ghostLiPositionFungibleStatesIndex64[ghostLiPositionFungibles[i]] == i + 1
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

❌Violated in `fungibleStatesPointToCorrectPosition`: https://prover.certora.com/output/52567/947c1892ded7436da9e1d5e4c7fbedc3/?anonymousKey=8589453499b0e0d0e078c932a50cae2638a36168

```
// VS-LI-14: Fungibles with state must be at their claimed position
// If a fungible has non-zero state, it must appear in the array at the position indicated
by its index
invariant fungibleStatesPointToCorrectPosition(env e)
    forall LicredityHarness.Fungible fungible.
        ghostLiPositionFungibleStates256[fungible] != 0
            => ghostLiPositionFungibles[ghostLiPositionFungibleStatesIndex64[fungible] - 1]
== fungible
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

Passed after the fix was applied.

✅ `fungiblesHaveCorrectIndex`: https://prover.certora.com/output/52567/eb37c1aa0f024039ae2193c3b1202eed/?anonymousKey=07550f6767f141b04dc9ae9ae85c8fb503747899

✅ `fungibleStatesPointToCorrectPosition`: https://prover.certora.com/output/52567/92f63783f54c40158efd001145fd6196/?anonymousKey=5e7f2f4af08898261b644f2381724e762f293d8b

[HIGH] `Licredity::decreaseDebtShare` bypasses interest accrual (#16)

A borrower can call `Licredity::decreaseDebtShare` to repay their loan. Because this call can only reduce debt, it's treated as "safe" and allowed outside the `Licredity::unlock` flow. However, interest accrues only during `unlock`, `swap`, and liquidity add/remove operations. Therefore calling `decreaseDebtShare` directly therefore uses the last `totalDebtBalance/totalDebtShare` without first accruing interest, so the repayment is computed from a stale state.

❌Violated in `transitionDebtChangesRequireInterestAccrual`: https://prover.certora.com/output/52567/ca0bbac3edc147a2919a43e86f164f18/?anonymousKey=0b7fbfb1c9559fe2caf98aec277dc0df5a7f3091

```
// ST-LI-02: Debt operations must collect interest when time has elapsed
// When any debt operation modifies totalDebtBalance or totalDebtShare,
// if time has passed since the last interest collection, the function must
// first collect interest by updating lastInterestCollectionTimestamp.
// This ensures borrowers cannot repay debt using stale ratios to avoid accrued interest.
rule transitionDebtChangesRequireInterestAccrual(
    env e, method f, calldataarg args
    ) filtered { f -> !EXCLUDED_FUNCTION(f) } {

    setupValidState(e);

    // Track state before operation
    mathint totalDebtBalanceBefore = ghostLiTotalDebtBalance128;
    mathint totalDebtShareBefore = ghostLiTotalDebtShare128;
    mathint lastCollectionBefore = ghostLiLastInterestCollectionTimestamp32;

    // Execute function
    f(e, args);

    // Check if a debt operation occurred (changes to debt balance or shares)
    bool debtOperationOccurred =
        ghostLiTotalDebtBalance128 != totalDebtBalanceBefore ||
        ghostLiTotalDebtShare128 != totalDebtShareBefore;

    // Check if time has elapsed since last interest collection
    bool timeElapsed = e.block.timestamp > lastCollectionBefore;

    // Verify: If debt changes AND time elapsed => interest must be collected (timestamp
updated)
    assert(debtOperationOccurred && timeElapsed => ghostLiLastInterestCollectionTimestamp32
== e.block.timestamp,
        "Debt operation occurred without collecting interest (timestamp not updated)");
```

```
}
```

Passed after the fix was applied.

✅ `transitionDebtChangesRequireInterestAccrual`: https://prover.certora.com/output/52567/06b9450bb741473eade3400b3260170b/?anonymousKey=cd99e1fe801d9d2bc42399c2751f492ae4b28f43

[INFO] Fungible array can contain zero-balance entries (#26)

The `addFungible` function allows adding fungibles with zero amounts, while `removeFungible` correctly removes fungibles when balance reaches zero. This creates an inconsistency where the fungibles array can legitimately contain entries with zero balance in `fungibleStates`.

❌Violated in `fungiblesHaveNonZeroBalance`: https://prover.certora.com/output/52567/a1c41658bd184bb39c3e91d196547010/?anonymousKey=21b3141cccc54343214d6d0e1ccb9791f40e1008

```
// VS-LI-11: Fungibles in array must have corresponding fungibleStates with non-zero balance
// For every fungible in the fungibles array, there must be a corresponding fungibleStates entry with non-zero balance
invariant fungiblesHaveNonZeroBalance(env e)
    forall mathint i.
        i >= 0 && i < ghostLiPositionFungiblesLength
            => ghostLiPositionFungibleStatesBalance112[ghostLiPositionFungibles[i]] != 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

[INFO] EIP-20 transfer functions accept zero address causing unintended token burns (#21)

EIP-20 compliance revealed that transfer() and transferFrom() accept the zero address as a recipient, which triggers token burning and causes unexpected totalSupply changes.

❌Violated in `eip20_transferFromIntegrity`, `eip20_transferFromMustRevert`, `eip20_transferIntegrity`, `eip20_transferMustRevert`: https://prover.certora.com/output/52567/9861c3c39be94927b7ff3e62d1c145ee/?anonymousKey=847508a55d39e425235808e6b07158d8324ee5dc

```
// EIP20-05: Verify transfer() reverts in invalid conditions
// EIP-20: "The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."
rule eip20_transferMustRevert(env e, address to, uint256 amount) {

    setup(e);

    // Snapshot the 'from' balance
    mathint fromBalancePrev = ghostERC20Balances128[_Licredity][e.msg.sender];

    // Attempt transfer with revert path
    transfer@withrevert(e, to, amount);
    bool reverted = lastReverted;

    assert(e.msg.sender == 0 => reverted,
           "[SAFETY] Transfer from zero address must revert");
```

```
    assert(to == 0 => reverted,
           "[SAFETY] Transfer to zero address must revert");

    assert(fromBalancePrev < amount => reverted,
           "[EIP-20] Transfer must revert if sender has insufficient balance");
}
```

❌Violated in `debtOutstandingWithinSupply`: https://prover.certora.com/output/52567/9e358de91dd34b63859d7707bc449ba9/?anonymousKey=0e965007f04aa90e4c70a60791e0326900d58f26

```
// VS-LI-05: Debt amount outstanding must not exceed total supply
// The debtAmountOutstanding represents debt tokens available for exchange back to base tokens.
// These are "free" tokens outside of positions that can be redeemed.
invariant debtOutstandingWithinSupply(env e)
    ghostLiDebtAmountOutstanding128 <= ghostERC20TotalSupply256[_Licredity]
filtered { f -> !EXCLUDED_FUNCTION(f)
    // SAFE: decreaseDebtShare burns debt tokens (reducing totalSupply) but doesn't affect debtAmountOutstanding
    && f.selector != sig:decreaseDebtShare(uint256,uint256,bool).selector
} { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

Passed after the fix was applied.

✅ `eip20_transferFromIntegrity`, `eip20_transferFromMustRevert`, `eip20_transferIntegrity`, `eip20_transferMustRevert`: https://prover.certora.com/output/52567/857699d6187d4d1492ec5ea5b465b8af/?anonymousKey=0b31ee944936f0fc97d5ea2d1c421de6220737a7

✅ `debtOutstandingWithinSupply`: https://prover.certora.com/output/52567/4af0dff6c89f47af8db76b3711d7c2af/?anonymousKey=9c397ce115ce3d469437eb924b97c3244c5a19c6

[INFO] Missing zero delta check in `increaseDebtShare` and `decreaseDebtShare` (#20)

The `Licredity::increaseDebtShare` and `Licredity::decreaseDebtShare` functions do not validate that the `delta` parameter is non-zero. A user can call these functions with `delta = 0`, which will result in an `amount` of 0. The function will proceed with its execution, performing all the necessary checks and state reads, but will ultimately not alter the position's debt or the total debt balance. However, it will still emit an `IncreaseDebtShare` or `DecreaseDebtShare` event with zero values.

❌Violated in `fungiblesHaveNonZeroBalance`: https://prover.certora.com/output/52567/a1c41658bd184bb39c3e91d196547010/?anonymousKey=21b3141cccc54343214d6d0e1ccb9791f40e1008

```
// VS-LI-11: Fungibles in array must have corresponding fungibleStates with non-zero balance
// For every fungible in the fungibles array, there must be a corresponding fungibleStates
entry with non-zero balance
invariant fungiblesHaveNonZeroBalance(env e)
    forall mathint i.
        i >= 0 && i < ghostLiPositionFungiblesLength
            => ghostLiPositionFungibleStatesBalance112[ghostLiPositionFungibles[i]] != 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

[INFO] Missing zero address check for `recipient` (#12)

The `Licredity::exchangeFungible`, `Licredity::withdrawFungible`, `Licredity::withdrawNonFungible`, and `Licredity::seize` functions does not validate the `recipient` address. For example, if a user calls `Licredity::withdrawFungible` and provides `address(0)` as the recipient, the function will proceed to transfer the fungible tokens to the zero address. Most token standards treat a transfer to the zero address as a burn, meaning the tokens are permanently and irretrievably lost.

❌Violated in `positionsWithDataMustHaveOwner`: https://prover.certora.com/output/52567/dba957329e054f1c814abc2fb09c5662/?anonymousKey=2897a7747ee78bcc0c7e519f14588431be170ef1

```
// VS-LI-03: Positions with any data must have an owner
// If a position has any data (debt, fungibles, or nonFungibles), it must have a non-zero
owner
invariant positionsWithDataMustHaveOwner(env e)
    ghostLiPositionDebtShare128 != 0 ||
    ghostLiPositionFungiblesLength != 0 ||
    ghostLiPositionNonFungiblesLength != 0
        => ghostLiPositionOwner != 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

❌Violated in `debtOutstandingWithinSupply`: https://prover.certora.com/output/52567/9e358de91dd34b63859d7707bc449ba9/?anonymousKey=0e965007f04aa90e4c70a60791e0326900d58f26

```
// VS-LI-05: Debt amount outstanding must not exceed total supply
// The debtAmountOutstanding represents debt tokens available for exchange back to base
tokens.
// These are "free" tokens outside of positions that can be redeemed.
invariant debtOutstandingWithinSupply(env e)
    ghostLiDebtAmountOutstanding128 <= ghostERC20TotalSupply256[_Licredity]
filtered { f -> !EXCLUDED_FUNCTION(f)
    // SAFE: decreaseDebtShare burns debt tokens (reducing totalSupply) but doesn't affect
debtAmountOutstanding
    && f.selector != sig:decreaseDebtShare(uint256,uint256,bool).selector
} { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

Passed after the fix was applied.

✅ `positionsWithDataMustHaveOwner`: https://prover.certora.com/output/52567/309e2706e1504c72bf99aee04ff4aa4f/?anonymousKey=1fde773a6132cecb80166fc8268083ab2cf02ad6

✅ `debtOutstandingWithinSupply`: https://prover.certora.com/output/52567/4af0dff6c89f47af8db76b3711d7c2af/?anonymousKey=9c397ce115ce3d469437eb924b97c3244c5a19c6

Setup and Execution Instructions

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

Required source code modifications

The following modifications are required in the source code for Certora verification:

1. **Licredity.sol**: Function visibility changes from `external` to `public`
    - Line 152: `stageFungible(Fungible fungible)`
    - Line 164: `exchangeFungible(address recipient, bool baseForDebt)`
    - Line 239: `depositFungible(uint256 positionId)`
    - Line 273: `withdrawFungible(uint256 positionId, address recipient, Fungible fungible, uint256 amount)`
    - Line 311: `stageNonFungible(NonFungible nonFungible)`
    - Line 327: `depositNonFungible(uint256 positionId)`
    - Line 369: `withdrawNonFungible(uint256 positionId, address recipient, NonFungible nonFungible)`
    - Line 407: `increaseDebtShare(uint256 positionId, uint256 delta, address recipient)`
    - Line 540: `seize(uint256 positionId, address recipient)`
    - Comment markers added: `// @certora external -> public`
    - These changes are necessary to allow the Certora harness to properly call and verify these functions
2. **FungibleState.sol**: Type modification
    - Line 7: Changed type definition from `bytes32` to `uint256`
    - Comment marker added: `// @certora bytes32 -> uint256`
    - This change ensures proper handling of the FungibleState type in the Certora verification environment

Verification Execution

The verification suite is organized by property categories, with configuration files in `certora/confs/`. You can run verifications for individual contracts or specific property types.

Running Verifications

1. **Valid State Invariants (Single Position)**:

    Verifies invariants for a single position scenario:

```
# Run all single position valid state invariants (19 properties)
certoraRun certora/confs/licredity_valid_state_single.conf

# Or use the helper script
cd certora/confs && ./run_valid_state_single.sh
```

2. **Valid State Invariants (Multi-Position)**:

   Verifies invariants across multiple positions:

```
# Run all multi-position valid state invariants (3 properties)
certoraRun certora/confs/licredity_valid_state_multi.conf
```

3. **State Transition Rules**:

   Verifies correct state transitions:

```
# Run state transition rules for single position
certoraRun certora/confs/licredity_state_transition_single.conf
```

4. **EIP-20 Compliance**:

   Verifies ERC-20 standard compliance:

```
# Run all EIP-20 compliance rules (11 properties)
certoraRun certora/confs/licredity_eip20_compliance.conf
```

Advanced Options

To optimize verification time or debug issues, you can run specific rules:

1. **Run with specific rule**:

```
# Run a specific invariant
certoraRun certora/confs/licredity_valid_state_single.conf --rule
positionsWithDataMustHaveOwner

# Run a specific EIP-20 rule
certoraRun certora/confs/licredity_eip20_compliance.conf --rule
eip20_transferIntegrity
```

2. **Run with specific method** (for parametric rules):

```
# Focus on a specific function
certoraRun certora/confs/licredity_state_transition_single.conf --method
"depositFungible(uint256)"
```