certora

Formal Verification Report

LIDO

Lido V3

December 2025

Prepared for Lido

Table of Contents

Project Summary

Project Scope

Project Name	Repository Link	Branch/PR/Commit Hash	Platform
Core	lidofinance/core/	Branch: #PR84 On-Chain,, commit 28571	On-Chain
Core	lidofinance/core/	Branch: #PR1287, Commit 2353e	On-Chain
Core	lidofinance/core/	Branch: #PR1287, Commit 15d69	On-Chain
Core	lidofinance/core/	Branch: #PR1467, Commit 86437	On-Chain
Core	lidofinance/core/	Branch: #PR1491, Commit 30106	On-Chain
Core	lidofinance/core/	Branch: #PR1512, Commit ebf90	On-Chain
Core	lidofinance/core/	Branch: #PR1528, Commit 32883	On-Chain
Core	lidofinance/core/	Branch: #PR1533, Commit 28b2f	On-Chain
Core	lidofinance/core/	Branch: #PR1541, Commit 0d3c6	On-Chain
Core	lidofinance/core/	Branch: #PR1549, Commit 7718c	On-Chain
Core	lidofinance/core/	Branch : #PR1568, Commit b983	On-Chain

Project Overview

This document describes the specification and verification of **Lido v3** using the Certora Prover. The work was undertaken from **June 20, 2025** to **December 4th, 2025**.

Last audited commit: `b98371488eb9479cf072bd6c2b682a59c5dd71d8`

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

Lido V3 introduces *Staking Vaults* (`stVaults`) as a modular extension to Lido's liquid staking protocol. The upgrade enables users to stake ETH with specific node operators while still minting stETH, preserving liquidity through an overcollateralization model.

Previously, stETH was minted only against ETH in the Lido Core pool. Lido V3 extends this to support external ETH backing through *StakingVault* contracts. Users deposit ETH into vaults and receive stETH based on a reserve ratio, e.g., 100 ETH deposits might mint 90 stETH at a 90% reserve ratio. The *VaultHub* contracts enforces that all stETH remains fully redeemable and maintains system-wide solvency.

Each ***StakingVault*** represents an isolated staking position controlled by a single owner and serviced by one node operator. Owners can deposit ETH to activate validators, connect to Lido via *VaultHub* to mint stETH against locked collateral, receive staking rewards, manage fee distribution, and trigger validator exits. Operators manage validators but cannot access deposited ETH.

The ***VaultHub*** is the central coordination layer of the protocol. It manages stETH minting and burning, enforces reserve ratio requirements based on each vault's *risk tier*. Vault owners can transfer ownership of the *StakingVaults* to the *VaultHub*, enabling them to mint stETH using explained reserve ratio mechanism. The *VaultHub* continuously monitors vault health through oracle reports. When vaults underperform or become undercollateralized, VaultHub can initiate rebalancing, forced disconnection, or bad debt socialization. Beyond this, the *VaultHub* is also responsible for coordinating redemption settlements, distributing fees to the protocol treasury, and tracking total locked value across all vaults to maintain protocol-wide invariants.

Individual vaults' *risk tiers* are managed by the ***OperatorGrid***. The contract organizes node operators into tiered groups, assigning distinct reserve ratios, share limits, forced rebalance thresholds, and fee structures. By default, a newly connected vault is assigned a default tier, but can transition to an operator-specific tier, provided a confirmation from both vault owner and the node operator.

The ***Predeposit Guarantee (PDG)*** mechanism protects against deposit frontrunning. Before executing a full validator deposit, the node operator precommits by calling the PDG contract with a 1 ETH bond, submitting the validator pubkey and BLS signature. The EIP-2537 BLS precompile verifies the signature on-chain, and withdrawal credentials are proven correct via

EIP-4788 beacon root proofs. After successful precommitment, the vault can deposit the remaining ETH (31 ETH for a standard validator, more for consolidation, top-ups). Only the precommitted pubkey can be used, preventing operators from frontrunning deposits. The guarantor receives their bond back after successful deposit execution.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	1	1	1
Medium	3	3	2
Low	5	5	3
Informational	1	1	0
Total	10	10	6

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
C-01	Locked value not updated when tier changes reserve requirements	Critical	Fixed
M-01	Redemptions can increase due to rounding in settlement	Medium	Fixed
M-02	Shortfall rebalance insufficient for vault health	Medium	Fixed
M-03	Staking limits bypass via external ether rebalancing	Medium	Acknowledged
L-01	Zero external shares despite non-zero vault liabilities due to rounding	Low	Fixed
L-02	If shares rate is less than 1 anyone can transfer shares from anyone else	Low	Acknowledged
L-03	Missing connection check allows operations on disconnected vaults	Low	Fixed
L-04	Solidity 0.4.24 allocates more memory than needed	Low	Acknowledged
L-05	Default tier params not validated during initialization	Low	Fixed
I-01	The value of fee shares may be too low	Informational	Acknowledged

Critical Severity Issues

C-01 - Locked Value Not Updated on Tier Change

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: vaultLockedCoversLiabilityAndReserve

Description:

`VaultHub.updateConnection()` fails to recalculate the vault's locked value when `OperatorGrid.changeTier()` modifies reserve ratio parameters. This allows vault owners to withdraw funds that should remain locked, violating health requirements and enabling undercollateralization.

VaultHub.sol (lines 474-479):

```
Python
function updateConnection(address _vault, uint256 _shareLimit, uint256 _reserveRatioBP,
uint256 _forcedRebalanceThresholdBP, uint256 _infraFeeBP, uint256 _liquidityFeeBP, uint256
_reservationFeeBP ) external onlyRole(CHANGE_CONNECTION_ROLE)
{
   // ... validation checks ...
   connection.shareLimit = uint96(_shareLimit);
   connection.reserveRatioBP = uint16(_reserveRatioBP);

   // Updated
   connection.forcedRebalanceThresholdBP = uint16(_forcedRebalanceThresholdBP);
   // Updated
   connection.infraFeeBP = uint16(_infraFeeBP);
   connection.liquidityFeeBP = uint16(_liquidityFeeBP);
   connection.reservationFeeBP = uint16(_reservationFeeBP);

   // Missing: record.locked recalculation
}
```

Attack scenario:

1. Vault has totalValue=1000 ETH, liabilityShares worth 369 ETH, locked=451 ETH, reserveRatioBP=10%
2. Vault is healthy under current parameters
3. OperatorGrid.changeTier() increases reserveRatioBP to 63.1% and forcedRebalanceThresholdBP to 20%
4. locked value remains 451 ETH despite higher reserve requirements
5. Owner immediately calls withdraw() for 548 ETH (totalValue - locked = 549 ETH available)
6. Withdrawal succeeds because locked value wasn't updated
7. Vault now unhealthy with insufficient collateral for new reserve ratio

The invariant "vault's locked value covers liability + reserve ratio" breaks immediately after tier changes.

Recommendations : Recalculate locked value in updateConnection() using the same formula as applyVaultReport(): locked = max(liability * TOTAL_BASIS_POINTS / (TOTAL_BASIS_POINTS - reserveRatioBP), CONNECT_DEPOSIT).

Lido's response: Fixed in #PR1425

Fix Review: updateConnection() now recalculates locked value when reserve parameters change.

Medium Severity Issues

M-01 - Redemptions can increase due to rounding in settlement

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: redemptionIncreaseViolation

Description:

In VaultHub's rebalancing mechanism, share-to-ETH conversion rounding causes redemption values to unintentionally increase during settlement operations. The issue occurs in the rebalancing calculation flow where shares are converted to ETH and back.

VaultHub.sol (lines 1324-1338):

```python
function _planRebalance( address _vault, VaultRecord storage _record, VaultObligations
storage _obligations ) internal view returns (uint256 valueToRebalance, uint256
sharesToRebalance)
{
   uint256 redemptionShares = _getSharesByPooledEth(_obligations.redemptions);
   uint256 maxRedemptionsValue = _getPooledEthBySharesRoundUp(redemptionShares);
   if (maxRedemptionsValue < _obligations.redemptions) redemptionShares += 1;

   uint256 cappedRedemptionsShares =
      Math256.min(_record.liabilityShares, redemptionShares);
   sharesToRebalance = Math256.min(cappedRedemptionsShares, _getSharesByPooledEth(balance));
   valueToRebalance = _getPooledEthBySharesRoundUp(sharesToRebalance);

}
```

The vulnerability chain:

1. `redemptionShares` calculated by converting ETH to shares (rounds down)
2. `maxRedemptionsValue` converts back to ETH using `_getPooledEthBySharesRoundUp` (rounds up)
3. If `maxRedemptionsValue < _obligations.redemptions`, an extra share is added
4. This process can cause small incremental increases in redemption values
5. Affected functions: `applyVaultReport()`, `resumeBeaconChainDeposits()`, and `settleVaultObligations()`

The rounding errors accumulate over multiple reports, gradually inflating redemption obligations beyond what should be settable only via `setVaultRedemptions()`.

Recommendations: Use consistent rounding direction (round down for both conversions) or explicitly cap redemption increases to prevent inflation outside of `setVaultRedemptions()`.

Lido's response: Fixed in PR #1350. The fix refactored the redemption calculation logic to use consistent rounding and prevent unintended increases.

Fix Review: The refactoring ensures redemptions can only increase through explicit setVaultRedemptions() calls by eliminating the rounding inconsistency in the conversion chain. The new implementation maintains redemption invariants across all settlement operations.

M-02 - Shortfall rebalance insufficient for vault health

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: shortfallValueIsSufficient

Description:

`VaultHub.rebalanceShortfall()` may leave a vault unhealthy due to rounding, especially when `reserveRatioBP == forcedRebalanceThresholdBP`.

Example: Vault has 1000 ETH totalValue, liability of 689 shares (801 ETH at current rate), RR=20%, and shortfall calculates as 5 shares. After rebalancing 5 shares (6 ETH worth), vault totalValue becomes 994 ETH, liability becomes 796 ETH, but threshold remains at 795 ETH. The vault is still unhealthy and requires multiple rebalance calls to achieve health.

Recommendations: Calculate the exact shortfall including a rounding buffer to ensure a single rebalance call is sufficient to restore vault health. Add `Math.ceilDiv()` for share-to-ETH conversions in shortfall calculations.

Lido's response: Fixed

Fix Review: Fixed in PR1549.

M-03 - Staking limits bypass via external ether rebalancing

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Lido.sol, VaultHub.sol	Status: Acknowledged	Violated Property: stakingLimitsAreKept

Description:

`Lido.rebalanceExternalEtherToInternal()` completely ignores staking limit mechanisms. This allows unlimited deposits via external rebalancing, circumventing `STAKING_CONTROL` and rate limiting. This breaks a fundamental protocol safeguard against excessive deposits.

Lido.sol

```python
function rebalanceExternalEtherToInternal(uint256 _amountOfShares) external
{
   _auth(_vaultHub()); _whenNotStopped();
   // Missing staking limit checks
   uint256 etherAmount = getPooledEthByShares(_amountOfShares);
   _burnShares(msg.sender, _amountOfShares);
   _setExternalShares(getExternalShares() - _amountOfShares);
   _submit(address(0));
}
```

Recommendations: Apply staking limit checks in the rebalance function, or exempt only amounts up to the current available limit. Ensure rebalancing operations respect global deposit limits.

Lido's response: Rebalancing bypasses the limits, but minting and burning external shares now takes the limits into account. Rebalancing does not affect the total issuance of a token (just internalizes shares) so it is safe from Lido perspective. It may also affect a simulatedShareRate check in OracleReportSanityChecker, but the parameter can be modified to avoid any interruption.

Fix Review: Acceptable on Certora side as rebalancing just internalizes shares.

Low Severity Issues

L-01 - Zero external shares despite non-zero vault liabilities due to rounding

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol, Lido.sol	Status: Fixed	Violated Property: strictlyTooManyLiabilityShares

Description:

When the internal ETH:shares ratio falls below 1, repeated small rebalances can burn more external shares than liability shares due to rounding up in share value calculations.

vaultHub.sol

```python
function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
   // Rounds up when converting shares to ETH value
   uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);
   // Burns external shares based on rounded-up value
}
```

Vulnerability scenario (when ratio is 35 ETH : 36 shares):

1. Vault has liabilities: 35 liability shares
2. Initial external shares: 36
3. Rebalance 35 shares: `_getPooledEthBySharesRoundUp(35) = 36 ETH` equivalent
4. Burns 36 external shares (all of them)
5. Liability shares still 35 (non-zero)
6. Result: externalShares = 0, liabilityShares = 35

This creates an accounting mismatch where the last vaults disconnecting could have non-zero liabilities but zero external shares backing them. The vault becomes unable to disconnect unless VaultHub manually internalizes the debt.

Recommendations: Use consistent rounding direction (round down for both share burns and liability calculations), or track and periodically reconcile rounding errors to prevent accumulation.

Lido's response: Fixed in #PR1419.

Fix Review: Fixed.

L-02 - If shares rate is less than 1 anyone can transfer shares from anyone else

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: StETH.sol	Status: Acknowledged	Violated Property: burnerDoesNotAffectThirdPartyShares

Description:

When the share rate falls below 1 ETH per share, anyone can call `transferSharesFrom()` without proper allowance if the ETH value of the shares being transferred rounds to zero.

StETH.sol (lines 388-396):

```python
function transferSharesFrom(
    address _sender, address _recipient, uint256 _sharesAmount
) external returns (uint256) {
    uint256 tokensAmount = getPooledEthByShares(_sharesAmount);
    // If tokensAmount rounds to 0, allowance check passes with 0
    _spendAllowance(_sender, msg.sender, tokensAmount);
    _transferShares(_sender, _recipient, _sharesAmount);
    return tokensAmount;
}
```

StETH.sol (lines 462-468):

```python
function _spendAllowance(address _owner, address _spender, uint256 _amount) internal {
    uint256 currentAllowance = allowances[_owner][_spender];
    if (currentAllowance != INFINITE_ALLOWANCE) {
        // If _amount is 0, this check passes with any allowance
        require(currentAllowance >= _amount, "ALLOWANCE_EXCEEDED");
        _approve(_owner, _spender, currentAllowance - _amount);
    }
}
```

Attack scenario (when share rate < 1):

1. Share rate: 0.9 ETH per share
2. Attacker calls transferSharesFrom(victim, attacker, 1) with 1 share
3. getPooledEthByShares(1) = 0 (rounds down)
4. _spendAllowance(victim, attacker, 0) passes (0 <= any allowance)
5. 1 share transferred without proper authorization

While this requires the extremely unlikely scenario of share rate < 1, it represents a logical flaw in access control.

Recommendations : Add explicit zero-value checks in both transferSharesFrom() and _spendAllowance() to prevent transfers when the token amount is zero.

Lido's response: Marked as duplicate of existing issue #796. While theoretically possible, the scenario of share rate < 1 is extremely unlikely in practice would require burning thousands of wei i gas to steal 1 wei of stETH this way.

Fix Review: Acknowledged.

L-03 - Missing connection check allows operations on disconnected vaults

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: updateOnlyConnected

Description:

The `settleVaultObligations()` function can be executed on disconnected vaults, allowing state modifications on vaults that are no longer tracked by VaultHub.

VaultHub.sol (line 911):

```python
function settleVaultObligations(address _vault) external
{
   // Missing: _requireConnected() check
   // Function proceeds to settle obligations even if vault disconnected
   VaultRecord storage record = _vaultRecord(_vault);
   VaultObligations storage obligations = _vaultObligations(_vault);
   _settleObligations(_vault, record, obligations, NO_UNSETTLED_ALLOWANCE);
}
```

Impact:

1. Disconnected vaults can have their obligations settled
2. State changes occur on vaults not actively managed by VaultHub
3. Creates inconsistent accounting between connected and disconnected vaults
4. Obligations updated for vaults that shouldn't be participating in protocol

This breaks the invariant that only connected vaults should have their obligations actively managed by the protocol.

Recommendation: Add `_requireConnected(_vault)` check at the beginning of `settleVaultObligations()` to ensure operations only affect active vaults.

Lido's response: Fixed by adding connection requirement check to the function.

Fix Review: The fix ensures obligation settlement only occurs for vaults actively connected to VaultHub, maintaining proper accounting boundaries..

L-04 - Solidity 0.4.24 allocates more memory than needed

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: deposit_contract.sol	Status: Acknowledged	Violated Property: N/A

Description:

A compiler bug in Solidity 0.4.24 causes excessive memory allocation for constant-size byte arrays. When creating `new bytes(48)`, the compiler allocates `32 * 48 = 1536` bytes instead of the intended 48 bytes.

SigningKeys.sol:

```python
function saveKeysSigs( uint256 _nodeOperatorId, uint256 _startIndex, uint256 _keysCount,
bytes _pubkeys, bytes _signatures ) external
{
   // Each signature is 96 bytes (BLS signature)
   // Code uses: new bytes(48) for pubkey allocation
   // Compiler bug: allocates 1536 bytes instead of 48
   bytes memory pubkey = new bytes(48);
   // ... signature processing
}
```

Impact:

- Every call to `saveKeysSigs` wastes approximately 1488 bytes of memory per key
- For batch operations with multiple keys, waste multiplies significantly

- Increased gas costs for all key signature save operations
- Affects all deployed contracts using Solidity 0.4.24

The bug occurs because Solidity 0.4.24 misinterprets constant array size declarations, treating the size as a multiplier rather than the total byte count.

Recommendations: Upgrade to a more recent solidity version.All subsequent key management operations will use correct memory allocation.

Lido's response: Acknowledged. The found problem is a optimization, although a tricky one and does not have any security impact, also, the mentioned library is used in NodeOperatorRegistry that is not touched by V3 update .The bug is noted and a fix is planned to be delivered with the next NodeOperatorRegistry update.

Fix Review: For existing deployed contracts, the inefficiency remains but is tolerable given the contract's limited usage patterns.

L-05 - Default tier params not validated during initialization

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: OperatorGrid.sol	Status: Fixed	Violated Property: tierReserveRatioGeThreshold

Description:

`OperatorGrid.initialize()` doesn't call `_validateParams()` for the default tier unlike `registerTiers()`. This means the default tier could have invalid parameters, such as `forcedRebalanceThresholdBP > reserveRatioBP`. This creates systemic risk since all new vaults start with potentially invalid parameters.

Recommendations: Add _validateParams() call for the default tier during initialization to ensure tier parameter validity from deployment.

Lido's response: Fixed in PR #1317. Default tier parameters now undergo validation during OperatorGrid initialization.

Fix Review: Fixed. Initialization now validates default tier parameters using the same validation logic applied to custom tiers, preventing invalid configurations.

Informational Issues

I-01 - The value of fee shares may be too low

Description:

Fee calculation logic may compute fee share values that are too low in certain edge cases, particularly when dealing with small liability amounts or specific rounding scenarios. The issue arises from the order of operations in fee calculations and share-to-ETH conversions.

When fees are calculated on small liability amounts or when multiple rounding operations compound, the resulting fee shares may round down more aggressively than intended, slightly shortchanging fee recipients. While individually small, over many operations this could accumulate to non-trivial amounts.

Recommendations: Review fee calculation order of operations to minimize precision loss.

Lido's response: Acknowledged. Fee calculations use conservative rounding that slightly favors the protocol over fee recipients.

Fix review: The behavior is acceptable.

Formal Verification

Verification Methodology

We performed verification of the **Lido** protocol using the Certora verification tool which is based on Satisfiability Modulo Theories (SMT). In short, the Certora verification tool works by compiling formal specifications written in the Certora Verification Language (CVL) and **Lido**'s implementation source code written in Solidity.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVL holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of a smart contract. We use constructors as a base case, and consider all other (relevant) externally callable functions that can change the storage as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective constructor. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after this computation any resulting state satisfies the invariant.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.
Manually audited and tested	In case of verification intractability, the rule is battle-tested with fuzz test and manual audit.

General Assumptions and Simplifications

Code Modifications (Munging)

The verification required targeted code modifications stored as git patches to enable tractable analysis. These modifications do not alter contract semantics:

- **`munge-total-shares-access.patch`**: Refactored internal access to total shares in `StETH.sol` by introducing a `_setTotalShares()` internal function. This enables the Prover to reason about share modifications without altering the contract's behavior.
- **`munge-strategy-lib.patch`**: Changed visibility of `MinFirstAllocationStrategy.allocate()` from `public` to `internal` to improve verification performance without affecting contract functionality.
- **`munge-IStakingVault-path-safe-assembly.patch`**: Modified safe assembly operations in staking vault interfaces to support static analysis.

Library and External Call Summarizations

Due to computational complexity and cryptographic operations that exceed SMT solver capabilities, the following libraries and external functions are summarized:

BLS Cryptography (BLS12_381 Library)

- `verifyDepositMessage()`, `sha256Pair()`, and `pubkeyRoot()` are summarized as `NONDET` (non-deterministic). The Prover cannot model BLS12-381 pairing operations and EIP-2537 precompiles, but assumes these cryptographic operations behave correctly when called.

SSZ and Consensus Layer Proofs

- `SSZ.hashTreeRoot()` and `SSZ.verifyProof()` summarized as `NONDET`
- `CLProofVerifier._validatePubKeyWCProof()` summarized as `NONDET`
- These summaries overapproximate beacon chain proof verification while maintaining soundness for the protocol logic being verified.

MerkleProof Library

- `MerkleProof.verify()` is summarized to always return `true` in LazyOracle properties, assuming Merkle proofs are correctly constructed when provided.

Share-ETH Conversion Functions

The following conversions between shares and ETH (implemented in `StETH.sol`) are summarized with simplified implementations:

- `getSharesByPooledEth(uint256)`
- `getPooledEthByShares(uint256)`
- `getPooledEthBySharesRoundUp(uint256)`

These summaries are functionally equivalent modulo arithmetic under/overflows and division by zero. In particular, conversions assume the denominator (total shares or total pooled ether) is non-zero, and all intermediate and final results fit within `uint256`.

Accounting Fee Calculations

- `Accounting._calculateTotalProtocolFeeShares()` is summarized with an equivalent implementation that simplifies arithmetic while assuming no overflows or division by zero occur. This enables tractable reasoning about fee distribution.

- `PositiveTokenRebaseLimiter.getSharesToBurnLimit()` is similarly summarized to avoid complex nested arithmetic.

Staking Router Assumptions

- The StakingRouter is assumed to have at most two staking modules with constant parameters throughout verification.
- Calls to `WithdrawalQueueBase.prefinalize()` are summarized assuming at least one share per batch and that maximum share rates are not exceeded.
- `StakingRouter.reportRewardsMinted()` and `StakingRouter.deposit()` are summarized as `NONDET`, which is not fully sound but acceptable for the properties verified.

Access Control and Governance

- OpenZeppelin's `AccessManager` framework permissions are summarized via `NONDET` for `hasPermission()` checks. The concrete access control configuration is post-deployment and outside verification scope.
- AragonOS `IKernel.hasPermission()` similarly summarized as `NONDET`.
- Recovery vault operations (`getRecoveryVault()`) are treated as non-deterministic.

Dispatcher Summaries

Several external interfaces use `DISPATCHER(true)` to allow the Prover to dispatch calls to their implementations when linking is available:

- StakingVault operations: `withdraw()`, `fund()`, `acceptOwnership()`, `requestValidatorExit()`, etc.
- PredepositGuarantee: `proveUnknownValidator()`
- Burner: `commitSharesToBurn()`, `requestBurnShares()`
- Vault operations: `withdrawRewards()`, `withdrawWithdrawals()`

Oracle and Timestamp Assumptions

- LazyOracle's `latestReportTimestamp()` is assumed constant (summarized as `CONSTANT` or `NONDET`) for properties that don't focus on timestamp progression.
- AccountingOracle's `getLastProcessingRefSlot()` is treated as constant, which is acceptable for the verified properties.
- Sanity checker functions (`checkAccountingOracleReport`, `checkWithdrawalQueueOracleReport`, etc.) use `DISPATCHER(true)`.

Optimistic Assumptions for Tractability

- **Optimistic Fallback**: Enabled to allow verification to proceed when certain operations cannot be fully resolved

- **Optimistic Hashing**: Hash operations use simplified modeling with `hashing_length_bound` of 98 bytes
- **Optimistic Loop**: Loop unrolling limited to 2 iterations (`loop_iter: 2`)

VaultHub and Staking Vaults

- Withdrawal credential proofs via EIP-4788 beacon roots are assumed correct (beacon block header validation summarized as `NONDET`)
- Triggerable withdrawals to the beacon chain withdrawal request contract are not fully modeled (`DISPATCH [] default NONDET`)

Lido and StETH

The following conversions between shares and ETH (implemented in `StETH.sol`) are summarized with simplified implementations:

- `getSharesByPooledEth(uint256)`
- `getPooledEthByShares(uint256)`
- `getPooledEthBySharesRoundUp(uint256)`

These summaries are functionally equivalent modulo arithmetic under- and overflows and division by zero errors. In particular, on any conversion from shares to ETH, or vice versa, assumes that the respective denominator or the conversion rate is non-zero, and the final result (as well as all intermediate calculations) fit into the desired type (`uint256`).

LazyOracle

- Properties assume no operations occur at block timestamp 0
- Quarantine state consistency rules rely on the assumption that quarantine periods and thresholds remain within reasonable bounds

Compilation Configuration

- Multiple Solidity versions are used: 0.4.24 (Lido core), 0.8.9 (Accounting, Burner), 0.8.25 (Vaults, OperatorGrid, PredepositGuarantee)
- EVM versions: `constantinople` for 0.4.24 contracts, `istanbul` for 0.8.9 contracts, `cancun` for 0.8.25 contracts
- Most contracts compiled without `via-ir`, except vault-related contracts (VaultHub, StakingVault, etc.) which use `via-ir` for optimization

All properties verified under these assumptions demonstrate correctness for the specified behavior, with explicit documentation where soundness is intentionally relaxed for tractability.

Formal Verification Properties Overview

ID	Title	Impact	Status
P-AC-01	Accounting Fee Computation	Ensures shares minted as fees represent their designated fraction of total rewards, and that fee shares are owned by the treasury, preventing fee miscalculation or misdirection.	Verified
P-AC-02	Oracle Report Handling Integrity	Ensures oracle reports can only be submitted by authorized oracle, with valid timestamps and validator counts, preventing unauthorized or malformed reports.	Verified
P-BU-01	Burner Shares Can Only Be Burnt	Ensures shares transferred to Burner can only be burned, never transferred to other accounts (except excess recovery), maintaining burn-only semantics.	Verified
P-BU-02	Burner Integrity	Ensures Burner operations do not affect unrelated parties' shares and that burn requests correctly move shares from sender to Burner, preserving isolation.	Verified
P-LI-01	Buffered ETH Backed by Balance	Ensures buffered ETH storage value is backed by actual native ETH balance of Lido contract, preventing accounting mismatches.	Verified
P-LI-02	Shares Transition	Verifies relations between external shares, internal shares, and total shares remain consistent, ensuring proper share accounting.	Verified
P-LI-03	Staking Limits Integrity	Ensures previous stake limit never exceeds maximum, block numbers are monotonically increasing, and staking limits remain	Verified

		unchanged during staking operations.	
P-LI-04	Staking Limits Enforcement	Verifies staking rate limits are enforced for all staking operations. Violated by `rebalanceExternalEtherToInternal()`	Verified
P-LI-05	Deposited Validators Monotonicity	Ensures deposited validator count is weakly monotonically increasing, preventing validator count manipulation.	Verified
P-LI-06	Share Conversion Summaries	Verifies summary implementations of `getSharesByPooledEth` and `getPooledEthBySharesRoundUp` match expected behavior.	Verified
P-LI-07	Staking Module Specific Operations	Ensures certain functions can only be called by Accounting and that rebalancing correctly adjusts internal/external ether accounting.	Verified
P-LI-08	Shares and Buffered ETH Changes	Shares transferred to the burner shall only be burned and never be transferred to another account.	Verified
P-NO-01	Node Operator Registry Integrity	Ensures the count of registered node operators only increases monotonically, preventing operator removal or registry corruption.	Verified
P-VH-01	Vault Array is Set	Ensures the `vaults` array in VaultHub contains no duplicate vault addresses, maintaining registry consistency.	Verified
P-VH-02	Vault Connection State Consistency	Verifies invariants relating vault connection status to liability shares and locked values: obligated vaults must be connected, disconnected vaults have zero liability/locked.	Verified

P-VH-03	Reserve Ratio Bounds	Ensures tier and vault reserve ratios never exceed 100% (≤ TOTAL_BASIS_POINTS) and that reserve ratios are greater than or equal to forced rebalance thresholds.	Verified
P-VH-04	Liability Accounting	Verifies that max liability shares are always ≥ current liability shares, redemption shares ≤ liability shares, and pending vaults have zero shares.	Verified
P-VH-05	Total Value and Redemption Monotonicity	Ensures vault total value can only increase through specific authorized functions, and that redemptions can only increase via setVaultRedemptions (catching Issue #1321).	Verified after fix
P-VH-06	Vault Health Preservation	Verifies that healthy vaults remain healthy until oracle reports are applied, ensuring health is preserved between reports under normal operations.	Verified
P-LO-01	Sanity Check Revert Conditions	Verifies revert conditions for `_handleSanityChecks`, ensuring sanity of incoming vault reports before updates are applied.	Verified
P-LO-02	Quarantine Soundness	Ensures quarantine state consistency: active quarantines have non-zero timestamps and pending values, quarantines cannot be reused after expiry, and funds are only released when quarantine ends.	Verified after Fix
P-PG-01	Validator Status State Machine	Ensures validator precommitment status follows valid state transitions (unknown → precommitted → deposited → proven), preventing reuse or invalid state changes.	Verified
P-OG-01	Immutable Ratio	Verifies that when reserve ratios are	Verified

	Health Invariant	immutable (not modified), vault locked values consistently cover liabilities and reserves, demonstrating that ratio changes are the cause of health violations.	
F-01	Locked Value Covers Liability and Reserve	Validates that vault locked values consistently cover liabilities and reserves across all state-changing operations including share rate changes, tier updates, rebalancing, and fee settlements, preventing undercollateralization under realistic conditions.	Manually audited and tested
F-02	Healthy Vault Preservation	Ensures healthy vaults remain healthy through arbitrary operation sequences until oracle reports are applied (with explicit exception for settleLidoFees), maintaining system stability between oracle updates.	Manually audited and tested
F-03	Shortfall Calculation Sufficiency	Verifies that rebalancing by `healthShortfallShares()` amount is sufficient to restore vault health across varying share rates and funding scenarios, ensuring the shortfall calculation is correct despite rounding (related to Issue #1305).	Manually audited and tested

Detailed Properties

Accounting

Module General Assumptions

- To ensure feasibility of formal verification, the following functions were summarized with respective functionally equivalent versions, that allow for simpler reasoning and additionally assume that neither arithmetic over- or underflows, nor divisions by zero occur:
  - `Accounting._calculateTotalProtocolFeeShares`
  - `PositiveTokenRebaseLimiter.getSharesToBurnLimit`
- We assume that the *StakingRouter* has at most two staking modules and the parameters of these modules are constant.
- Calls to `WithdrawalQueueBase.prefinalize` are simplified with a summary that only assumes that there is at least one share to burn per batch and that the maximal share rate is not surpassed.

Module Properties

P-AC-01. Accounting Fee Computation.	
Status: Verified	Integrity rules of the Accounting contract.

Rule Name	Status	Description	Link to rule report
feesAreFraction	Verified	*The value of the shares minted as fees is roughly their designated fraction* `(toMintAsFees * update.postInternalEther) / postInternalShares)` *of the total rewards. In particular* 1. *fee shares are not worth more than the designated fraction rounded down, and* 2. *fee shares value rounded up are not worth less than half designated fraction.*	*Report*

		Note that it was acknowledged (#1457) that fee shares value rounded up is indeed allowed to be worth less than designated fraction.	
feesMintShares	Verified	*When shares are minted as fees, then balance increase does not exceed the rewards from the oracle report and the recipient of the shares is the treasury.*	*Report*

P-AC-02. Oracle Report Handling Integrity.

Status: Verified	Integrity of `handleOracleReport`

Rule Name	Status	Description	Link to rule report
handleOracleReportRevertConditions	Verified	*A call to `handleOracleReport` reverts if either* • *the sender is not the AccountingOracle* • *the report timestamp larger than or equal to the current block's timestamp,* • *the report's number validators is less than the current number of consensus layer validators, or* • *the report's number of validators is larger than then current number of deposited validators.*	*Report*

Burner

Module General Assumptions

- Share Rate ≥ 1: Assumes one share is worth at least one ETH to avoid known edge case violations when share rate drops below 1 (Issues #1399 and #796).
- Summarized Share Conversions: Uses simplified CVL implementations of Lido's share-to-ETH conversion functions (`CVLgetPooledEthByShares`, `CVLgetSharesByPooledEth`) for tractable verification.
- Filtered Functions: Excludes `recoverERC20()` and `recoverERC721()` from verification as they interact with external contracts via safeTransfer, which is outside the core Burner security model.

Module Properties

P-BU-01. Burner shares can only be burnt, never transferred.	
Status: Verified	Shares transferred to the burner shall only be burned and never be transferred to another account.

Rule Name	Status	Description	Link to rule report
burnerDoesNotApprove	Verified	*The `Burner` contract gives no allowance to any address.*	*Report*
burnerSharesOnlyBurnt	Verified	*If the number of shares owned by the `Burner` contract decreases, then only because they were burned. The only exception to this is recovering excessive stETH by calling `recoverExcessStETH()`.*	*Report*

P-BU-02. Burner Integrity.

Status: Verified	Ensures Burner operations do not affect unrelated parties' shares and that burn requests correctly move shares from sender to Burner, preserving isolation.

Rule Name	Status	Description	Link to rule report
burnerDoesNotAffectThirdPartyShares	Verified	*Burner does not affect unrelated parties' shares. In particular, if a function on the `Burner` contract is called, then only the shares of either the contract itself or the share of the caller can change.* *Note: We assume that one share is worth at least one ETH. Otherwise the property does not hold due to an acknowledged issue requiring catastrophic conditions (see #1399 and #796) .*	*Report*
burnRequestsIntegrity	Verified	*Calling either of the five burn functions increases the `Burner` shares, decreases the caller's shares, and leaves the shares of any other address untouched. This assumes that the `Burner` contract does not invoke the burn functions itself.*	*Report*
commitBurnIntergrity	Verified	*Ensures `commitSharesToBurn()` burns precisely the requested share amount from total supply and exclusively from the Burner's own balance, preventing burns from affecting other accounts.*	*Report*

Lido Core

Module General Assumptions

- Share-to-ETH Conversion Summaries: The functions `getSharesByPooledEth()` and `getPooledEthBySharesRoundUp()` are replaced with CVL summary implementations (lines 128-172) that simplify multiplication/division operations while assuming: (1) numerator/denominator are non-zero to avoid division by zero, (2) values fit within uint128 bounds to prevent overflows, and (3) no overflow in intermediate calculations.
- External Contract Summaries: `StakingRouter`, `WithdrawalQueue`, `Burner`, `Accounting`, and `VaultHub` operations are summarized as NONDET or `DISPATCHER(true)`. This includes `deposit(), reportRewardsMinted(), finalize()`, and burn operations, which are not fully modeled but assumed to behave correctly.
- Reasonable Value Bounds: Rules assume *maxReasonableValue() = 2^100* for shares, buffered ETH, and ETH transfer values to prevent arithmetic overflows during verification while maintaining realistic bounds.

Module Properties

P-LI-01. Buffered ETH Backed by Balance.

Status: Verified	Ensures buffered ETH storage value is backed by actual native ETH balance of Lido contract, preventing accounting mismatches.

Rule Name	Status	Description	Link to rule report
bufferedEthBackedByBalance	Verified	*Ensures `bufferedEther` storage value ≤ native ETH balance of Lido contract*	*Report*

P-LI-02. Shares Transition.

Status: Verified	Verifies relations between external shares, internal shares, and total shares remain consistent, ensuring proper share accounting.

Rule Name	Status	Description	Link to rule report
sharesTransition	Verified	Ensures consistent share accounting by verifying the relationship between external, internal, and total shares.	*Report*

P-LI-03. Staking Limits Integrity.

Status: Verified	Ensures previous stake limit never exceeds maximum, block numbers are monotonically increasing, and staking limits remain unchanged during staking operations.

Rule Name	Status	Description	Link to rule report
prevStakingBlockNumberIncreasing	Verified	*Ensures previous staking block number is weakly monotonically increasing*	*Report*
stakingLimitsUnchangedIfStaking	Verified	*Ensures max stake limit and growth blocks remain unchanged during staking operations*	*Report*

P-LI-04. Staking Limits Enforcement.

Status: Verified	Verifies staking rate limits are enforced for all staking operations.

Rule Name	Status	Description	Link to rule report
StakingLimitsAreKept	Verified	*Verifies staking rate limits are enforced, specific scenario for rebalancing external shares acknowledged by Lido's team that won't be fixed violated the issue.*	*Report*

P-LI-05. Deposited Validators Monotonicity.

Status: Verified	Ensures deposited validator count is weakly monotonically increasing, preventing validator count manipulation.

Rule Name	Status	Description	Link to rule report
depositedValidatorsOnlyIncreasing	Verified	*Ensures deposited validator count is weakly monotonically increasing*	*Report*

P-LI-06. Shares Conversion Summaries

Status: Verified	Ensures summary of CVL function matches expected behavior.

Rule Name	Status	Description	Link to rule report
verifygetSharesByPooledEthSummary	Verified	*Verifies CVL summary of getSharesByPooledEth*	*Report*
verifygetPooledEthBySharesRoundUp	Verified	*Verifies CVL summary of getPooledEthBySharesRoundUp matches implementation*	*Report*

P-LI-07. Accounting access control

Status: Verified	Ensures certain functions can only be called by Accounting and that rebalancing correctly adjusts internal/external ether accounting.

Rule Name	Status	Description	Link to rule report
canOnlyBeCalledByAccounting	Verified	*Ensures collectRewardsAndProcessWithdrawals() can only be called by the Accounting contract*	*Report*

P-LI-08. Shares and Buffered ETH Changes.

Status: Verified	Ensure shares and buffered ETH can only be changed by known functions.

Rule Name	Status	Description	Link to rule report
totalSharesCanOnlyBeChangedBy	Verified	*When total shares increase, must be via: submit(), mintShares(),mintExternalShares(),fallback,initialize(),or Accounting contractWhen total shares decrease, must be via: burnShares(), burnExternalShares(), or Accounting contractFilters out: deprecated functions, view functions, upgrade functions*	Report
bufferedEthCanOnlyBeChangedBy	Verified	*When buffered ETH changes, must be via : deposit(), submit(),mintShares(),rebalanceExternalEtherToInternal(),collectRewardsAndProcessWithdrawals(), initialize(), fallback, or Accounting contractFilters out: deprecated functions, view functions, upgrade functions*	Report

Node Operator

Module General Assumptions

- SigningKeys Library Summarization: The `SigningKeys` library functions (`initKeysSigsBuf()` and `loadKeysSigs()`) are summarized as `NONDET` or simplified implementations to avoid pointer analysis failures and complex memory operations, while maintaining correct buffer length constraints (PUBKEY_LENGTH = 48 bytes).
- AragonOS Access Control: `IKernel.hasPermission()` and `ConversionHelpers.dangerouslyCastUintArrayToBytes()` from AragonOS are summarized as `NONDET`, as access control configuration is post-deployment and the conversion helper only feeds into the non-deterministic permission check.
- Code Modification Required: Verification requires the `munge-strategy-lib.patch` to change `MinFirstAllocationStrategy.allocate()` visibility from `public` to `internal`, preventing the function from havocing the main contract state during symbolic execution.

Module Properties

P-NO-01. Node Operator Monotonicity	
Status: Verified	Ensures the count of registered node operators only increases monotonically, preventing operator removal or registry corruption.

Rule Name	Status	Description	Link to rule report
OperatorsCountIsIncreasing	Verified	*Ensures node operator count is weakly monotonically increasing and increases by at most 1 per transaction, preventing bulk operator removal or registry corruption.*	*Report*

VaultHub

Module General Assumptions

- **Share-to-ETH Conversion Summaries**: Uses CVL implementations of `getPooledEthBySharesRoundUp()` and `getSharesByPooledEth()` that assume non-zero denominators, values within uint128 bounds, and no intermediate overflows, matching the Lido base assumptions.
- **External Contract Interactions**: BLS cryptography (verifyDepositMessage, pubkeyRoot), SSZ proofs (hashTreeRoot, verifyProof), and beacon chain proof verification (_validatePubKeyWCProof) are all summarized as NONDET, as cryptographic verification exceeds SMT solver capabilities.
- **Reasonable Value Bounds**: Assumes `maxReasonableValue() = 2^100` for vault total values, deltas, and share amounts to prevent arithmetic overflows while maintaining realistic bounds for verification tractability.
- **Initialization Assumption**: All invariants assume `VaultHub.initialize()` is called immediately after the constructor, ensuring the vaults array is properly initialized with address(0) at index 0.

Module Properties

P-VH-01. Vault Array is Set	
Status: Verified	Ensures the `vaults` array in VaultHub contains no duplicate vault addresses, maintaining registry consistency.

Rule Name	Status	Description	Link to rule report
disconnectedVaultIsNotPending	Verified	*Ensures a vault pending disconnect still has non-zero index (remains connected), preventing premature state corruption.*	*Report*

indexToVaultIsCorrect	Verified	*Verifies `connections[vaults[i]].vaultIndex == i` for all valid indices, ensuring forward mapping correctness.*	*Report*
vaultToIndexIsCorrect	Verified	*Verifies `vaults[connections[v].vaultIndex] == v` for all vaults, ensuring reverse mapping correctness and that vaults array forms a proper set.*	*Report*

P-VH-02. Vault Connection State Consistency

Status: Verified	Verifies invariants relating vault connection status to liability shares and locked values: obligated vaults must be connected, disconnected vaults have zero liability/locked.

Rule Name	Status	Description	Link to rule report
obligatedVaultIsConnected	Verified	*Ensures vaults with obligation shares or unsettled Lido fees must be connected, preventing orphaned obligations.*	*Report*
disconnectedVaultHasNoLiability	Verified	*Ensures disconnected vaults have zero liability shares, maintaining clean state separation.*	*Report*
disconnectedVaultHasNoLocked	Verified	*Ensures disconnected vaults have zero locked value, preventing locked funds in disconnected vaults.*	*Report*

P-VH-03. Reserve Ratio Bounds

Status: Verified	Ensures tier and vault reserve ratios never exceed 100% (≤ TOTAL_BASIS_POINTS) and that reserve ratios are greater than or equal to forced rebalance thresholds.

Rule Name	Status	Description	Link to rule report
tierReserveRatioLeqOne	Verified	*Ensures each tier's reserve ratio ≤ MAX_RESERVE_RATIO_BP (9999), preventing invalid tier configurations.*	*Report*
reserveRatioNotBig	Verified	*Ensures vault reserve ratios ≤ TOTAL_BASIS_POINTS (10000), preventing ratios exceeding 100%.*	*Report*
tierReserveRatioGeThreshold	Verified	*Ensures tier reserve ratio > forced rebalance threshold, catching Issue #1291.*	*Report*
vaultReserveRatioGeThreshold	Verified	*Ensures vault reserve ratio > forced rebalance threshold for all connected vaults, preventing invalid health calculations.*	*Report*

P-VH-04. Liability Accounting

Status: Verified	Verifies that max liability shares are always ≥ current liability shares, redemption shares ≤ liability shares, and pending vaults have zero shares.

Rule Name	Status	Description	Link to rule report
maxLiabilitySharesGeqLiabilityShares	Verified	*Ensures max liability shares ≥ current liability shares for all vaults, preventing accounting inconsistencies.*	*Report*

redemptionSharesLeqLiabilityShares	Verified	*Ensures redemption shares ≤ liability shares, preventing over-redemption.*	*Report*
pendingHasNoShares	Verified	*Ensures vaults pending disconnect have zero liability and obligation shares, maintaining clean disconnect state.*	*Report*

P-VH-05. Total Value And Redemption Monotonicity

Status: Verified After Fix	Ensures vault total value can only increase through specific authorized functions, and that redemptions can only increase via `setLiabilitySharesTarget`.

Rule Name	Status	Description	Link to rule report
canIncreaseTotalValue	Verified after fix	*Ensures vault total value only increases via fund(), applyVaultReport(), or connectVault()*	*Report*
redemptionsIncrease	Verified after fix	*Ensures unsettled Lido redemptions only increase via applyVaultReport()*	*Report*

P-VH-06. Vault Health Preservation

Status: Verified	Verifies that healthy vaults remain healthy until oracle reports are applied, ensuring health is preserved between reports under normal operations.

Rule Name	Status	Description	Link to rule report

vaultIsHealthyUntilReport	Verified	*Ensures healthy vaults remain healthy until applyVaultReport() is called (excluding settle fees), assuming share rate ≥ 1, previously violated by rounding in rebalance functions.*	Report
summaryCorrect	Verified	*Verifies CVL summary implementation of _withdrawableValueFeesIncluded matches expected behavior.*	Report

Lazy Oracle

Module General Assumptions

- **MerkleProof Always Valid**: `MerkleProof.verify()` is summarized to always return true, assuming Merkle proofs for vault data are correctly constructed when submitted.
- **Reasonable Value Bounds**: Uses `maxReasonableValue() = 2^100` for quarantine pending values and vault deltas to prevent arithmetic overflows while maintaining realistic bounds.
- **No Operations at Timestamp Zero**: All quarantine-related properties assume no operations occur at block timestamp 0, which is a reasonable assumption for mainnet.

Module Properties

P–LO-01. Sanity Check Revert Conditions	
Status: Verified	Verifies the revert conditions for `_handleSanityChecks`, which ensure the sanity of incoming reports before a vault is updated.

Rule Name	Status	Description	Link to rule report
handleSanityChecksRevertConditions	Verified	*Verifies all revert conditions for `_handleSanityChecks` including: underflow checks when deltas are applied, total value exceeding uint96, onchain value calculation overflows, cumulative Lido fees validation (must not decrease and rate must not exceed max), and max liability shares validation (must be ≥ liability shares and not exceed previous max).*	*Report*

P-LO-02. Quarantine Soundness

Status: Verified	Ensures quarantine state consistency: active quarantines have non-zero timestamps and pending values, quarantines cannot be reused after expiry, and funds are only released when quarantine ends.

Rule Name	Status	Description	Link to rule report
quarantineIntegrity	Verified	*Ensures quarantine remains active until end time, funds are not released while quarantine is active, funds are released after quarantine ends, and total value never exceeds reported value.*	*Report*
quarantineStateConsistency	Verified	*Ensures active quarantines have non-zero start timestamp and pending value increase (with valid end time > start time if period is non-zero), and inactive quarantines have all fields zeroed.*	*Report*
quarantineExpiry	Verified after Fix	*Ensures expired quarantines cannot be reused (current quarantine must expire or a new one with current timestamp must be created), previously violated - Issue #1304.*	*Report*

Predeposit Guarantee

Module General Assumptions

- **BLS and Cryptographic Summaries**: BLS12-381 operations (`verifyDepositMessage, sha256Pair, pubkeyRoot`) and SSZ operations (`hashTreeRoot, verifyProof`) are summarized as NONDET, as EIP-2537 BLS precompile verification and Merkle proof verification exceed SMT solver capabilities.
- **Beacon Chain Proof Verification**: `CLProofVerifier._validatePubKeyWCProof()` summarized as NONDET, assuming EIP-4788 beacon root proofs for validator withdrawal credentials are correctly validated when provided.
- **Deposit Root Calculation**: `_depositDataRootWithZeroSig()` summarized as NONDET to avoid sanity problems, assuming deposit data root computation is correct.
- **Beacon Chain Deposit**: `depositToBeaconChain()` summarized as NONDET (not fully sound), as interaction with the actual beacon chain deposit contract is external to the protocol logic being verified.

Module Properties

P-PG-01. Validator Status State Machine	
Status: Verified	Ensures validator precommitment status follows valid state transitions (none → predeposited → proven → activated), preventing reuse or invalid state changes.

Rule Name	Status	Description	Link to rule report
validatorStatusTransitions	Verified	*Ensures validator status follows valid state machine transitions*	*Report*

Operator Grid

Module General Assumptions

- **Reserve Ratios Remain Constant:** The `vaultLockedCoversLiabilityAndReserveImmutableRatio` invariant filters to only `rebalance(), forceRebalance(), and resumeBeaconChainDeposits()`. It excludes OperatorGrid functions that modify reserve ratios (`changeTier(), syncTier(), updateConnection()`), assuming ratios do not change during verification.
- **Pre-existing Ratio Validity:** The invariant's preserved block requires vaultReserveRatioNotGreaterThanThreshold(vault), assuming the vault's reserve ratio ≥ forced rebalance threshold before the rebalancing operations. This assumes OperatorGrid has already configured valid ratios.
- **Tier Configuration Stability:** The spec uses numTiers() and tier(tierId) from OperatorGrid as envfree (read-only), assuming tier configurations are stable and not modified during the verification of rebalancing operations.

Module Properties

P-OG-01. Immutable Ratio Health Invariant	
Status: Verified	Verifies that when reserve ratios are immutable (not modified), vault locked values consistently cover liabilities and reserves, demonstrating that ratio changes are the cause of health violations.

Rule Name	Status	Description	Link to rule report
vaultLockedCoversLiabilityAndReserveImmutableRatio	Verified	*Proves that when share-to-ETH conversion rate is immutable (constant), locked value consistently covers liability + reserve, demonstrating that violations of the standard health invariant are caused by conversion rate changes rather than rebalancing logic.*	*Report*

Testing and Fuzzing

Module General Assumptions

- **Mock StETH Share Rate Control**: Uses `MockStETH.setShareRateBP()` and `simulateRebalanceRateIncrease()` to simulate share rate changes, assuming the mock accurately reflects Lido's share-to-ETH conversion behavior including rounding (round-up for `getPooledEthBySharesRoundUp`).
- **Oracle Report Simulation**: Assumes `_applyReport()` and `_applyReportForVault()` correctly simulate LazyOracle's `applyVaultReport()` behavior, including timestamp progression, vault record synchronization, and inOutDelta tracking, without requiring actual oracle consensus or Merkle proofs.
- **Health-Preserving Tier Changes**: Tests for `changeTier() and updateConnection()` only proceed when the vault would remain healthy with new parameters (via `_isThresholdBreached()` check), assuming operators won't intentionally break health, focusing verification on parameter changes that preserve health.
- **Handler-Based Invariant Testing:** Uses HealthHandler contract as the target for Foundry's invariant testing framework, assuming the handler's operations (fund, mintShares, rebalance, etc.) accurately represent all possible state transitions that should preserve vault health
- **Unhealthy Initial State:** Tests assume the vault starts unhealthy (created in `setUp()` by minting shares and increasing share rate to 1.15x), allowing verification of shortfall calculation and recovery mechanisms.
- **Oracle Report Application:** Assumes `_applyReport()` correctly simulates LazyOracle's `applyVaultReport()` behavior, including timestamp updates and vault record synchronization, without requiring actual oracle consensus.

F-01. Locked Value Covers Liability and Reserve

Status: Manually audited and tested	Validates that vault locked values consistently cover liabilities and reserves across all state-changing operations including share rate changes, tier updates, rebalancing, and fee settlements, preventing undercollateralization under realistic conditions.

Rule Name	Status	Description
invariant_lockedCoversLiabilityAndReserve	Manually audited and tested	*Verifies the invariant locked (TOTAL_BASIS_POINTS - reserveRatioBP) >= liabilityEth * TOTAL_BASIS_POINTS holds after fund(),mintshares(),withdraw(),rebalance(),burnShares(),shareIncrease(),updateConnection() ,new liability target, settleFees()*

F-02. Healthy Vault Preservation

Status: Manually audited and tested	Ensures healthy vaults remain healthy through arbitrary operation sequences until oracle reports are applied (with explicit exception for settleLidoFees), maintaining system stability between oracle updates.

Rule Name	Status	Description
invariant_healthyVaultRemainsHealthy	Manually audited and tested	*A healthy vault remains healthy after any operation, verifying health preservation across all state transitions.*
testFuzz_healthyVaultRemainsHealthy	Manually audited and tested	*Fuzz test verifying the invariant through randomized operations (fund, mint, withdraw, rebalance, forceRebalance, burn, pause/resume, updateConnection, setLiabilityTarget), ensuring health preservation across operation sequences.*

F-03. Shortfall Calculation Sufficiency

Status: Manually audited and tested	Verifies that rebalancing by `healthShortfallShares()` amount is sufficient to restore vault health across varying share rates and funding scenarios, ensuring the shortfall calculation is correct despite rounding.

Rule Name	Status	Description	Link to rule report
testFuzz_shortfallOnInitialUnhealthyState	Manually audited and tested	*Verifies that rebalancing by healthShortfallShares() restores vault health when starting from an unhealthy state, across varying extra funding amounts.*	*N/A*
testFuzz_shortfallWithShareRateVariation	Manually audited and tested	*Verifies that shortfall calculation and rebalancing restore vault health across share rate variations (1.1x to 1.3x) and additional funding scenarios.*	*N/A*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.