Formal Verification Report: CircuitBreaker

- Date: April 6, 2026
- Audit Repo: https://github.com/Cyfrin/audit-2026-04-lido-circuit-breaker
- Client Repo: https://github.com/lidofinance/circuit-breaker
- Audit Commit: a9e4d1e30fafaaa4162029fe819833b35b21da0c
- Mitigation Commit: 30b01f13792e73b4dfc50e4aa093ab4dbf36802a
- Author: Bastion v2.3.6 created by Dacian
- Certora Prover version: 8.8.1

Table of Contents

About CircuitBreaker

`CircuitBreaker` is an emergency pause manager for the Lido protocol. It allows a trusted admin (Lido DAO governance agent) to delegate pause authority to designated multisig committees (pausers) that can act instantly in emergencies. Each pausable contract has one assigned pauser. Pauses are single-use (the pauser is unregistered after triggering), and pausers must maintain liveness via periodic heartbeats.

The `Registry` library maintains an enumerable set of pausable-to-pauser mappings using a swap-and-pop pattern for O(1) additions and removals, with bidirectional index tracking and per-pauser count aggregation.

Formal Verification Methodology

Certora Formal Verification (FV) provides mathematical proofs of smart contract correctness by verifying code against a formal specification. Unlike testing and fuzzing which examine specific execution paths, Certora FV examines all possible states and execution paths.

The process involves crafting properties in CVL (Certora Verification Language) and submitting them alongside compiled Solidity smart contracts to a remote prover. The prover transforms the contract bytecode and rules into a mathematical model and determines the validity of rules.

Types of Properties

Invariants — System-wide properties that MUST always hold true. These are parametric — automatically verified against every external function in the contract. Once proven, invariants serve as trusted assumptions via `requireInvariant`.

Parametric Rules — Rules verified against every non-view external function using `method f` with `calldataarg args`. Used for properties like "only admin can change pauseDuration" or "registry state only changes via registerPauser or pause."

Access Control Rules — Parametric state-change rules verifying that only authorized callers can modify critical state variables. For each protected variable, a parametric rule snapshots state before and after every function call, then asserts: if state changed, the caller must hold the required role. This is strictly stronger than function-specific `@withrevert` testing because it catches unauthorized modifications by unexpected functions.

Revert Condition Rules — Rules verifying that functions revert under specific invalid conditions (non-admin caller, expired heartbeat, unauthorized pauser, zero pausable address). Uses `@withrevert` followed by `assert lastReverted`.

Integrity Rules — Rules verifying that successful function calls produce the correct state changes (e.g., heartbeat sets expiry to `block.timestamp + heartbeatInterval`, pause clears the pauser mapping, idempotent unregistration is a net no-op).

Reachability Rules — Targeted `satisfy` rules proving specific scenarios are reachable (e.g., a registered live pauser can successfully call `pause`). General vacuity checking is handled by `rule_sanity: "basic"` in the conf file.

Key modeling decisions

- **IPausable summarization**: External calls to `pauseFor(uint256)` and `isPaused()` are summarized as `NONDET` since the pausable contract implementations are out of scope and no properties depend on cross-call consistency
- **Transient storage**: EVM version set to `cancun` for EIP-1153 transient storage support (reentrancy guard `bool internal transient lock`)
- **Ghost variables**: Persistent ghost tracking `pausableCount` sum via Sstore/Sload hooks for aggregation invariants. Persistent ghosts survive havoc from unresolved external calls
- **Harness**: `CircuitBreakerHarness` exposes three internal `Registry.Storage` fields (`oneBasedIndex`, `pausables[i]`, `pausables.length`) that are not accessible through `CircuitBreaker`'s public interface
- **Filtered invariants**: Three structural invariants (`membershipEquivalence`, `cleanStateAfterRemoval`, `globalCountConservation`) use `filtered` clauses to exclude `registerPauser` from their inductive step, with compensating per-function rules covering the excluded paths. This breaks a circular inductive dependency between enumerable set invariants (see Known Limitations in Verification Results)

Project Structure

```
certora/
├── conf/
│   └── CircuitBreaker.conf
├── harness/
│   └── CircuitBreakerHarness.sol
├── spec/
│   └── CircuitBreaker.spec
├── invariants.md
├── README.md
└── README.pdf
```

- `conf/` — Prover configuration files
- `harness/` — `CircuitBreakerHarness` inherits `CircuitBreaker` and adds three public getters for internal Registry state: `getOneBasedIndex`, `getPausableAt`, `getPausablesLength`
- `spec/` — CVL specification files containing all 41 rules
- `invariants.md` — 30 consolidated invariants from the discovery phase

Verification Properties

41 properties across 1 contract (10 invariants, 11 parametric rules, 4 access control rules, 6 revert condition rules, 7 integrity rules, 3 reachability rules).

CircuitBreaker - Registry Enumerable Set Integrity

ID	Name	Type	Description	Audit	Mitig.
CB-1	`forwardIndexConsistency`	Invariant	For every valid index `i`: `oneBasedIndex[pausables[i]] == i + 1`	✓	✓
CB-2	`reverseIndexConsistency`	Invariant	For every `p` with `oneBasedIndex[p] != 0`: `pausables[oneBasedIndex[p] - 1] == p`	✓	✓
CB-3	`membershipEquivalence`	Invariant	`pauser[p] != 0 <=> oneBasedIndex[p] != 0` (filtered; see CB-3a, CB-3b)	✓	✓
CB-3a	`membershipEquivalence_registerPauser`	Integrity	Membership equivalence preserved by `registerPauser`	✗	✗
CB-3b	`membershipEquivalence_pause`	Integrity	Membership equivalence preserved by `pause`	✓	✓
CB-4	`arrayEntriesHavePauser`	Invariant	For every valid index `i`: `pauser[pausables[i]] != 0`	✓	✓
CB-5	`noDuplicates`	Integrity	For all valid indices `i != j`: `pausables[i] != pausables[j]`	✓	✓
CB-6	`cleanStateAfterRemoval`	Invariant	`oneBasedIndex[p] == 0 => pauser[p] == 0` (filtered; see CB-6a, CB-6b)	✓	✓

ID	Name	Type	Description	Audit	Mitig.
CB-6a	`cleanStateAfterRemoval_registerPauser`	Integrity	Clean state preserved by `registerPauser`	✗	✗
CB-6b	`cleanStateAfterRemoval_pause`	Integrity	Clean state preserved by `pause`	✓	✓
CB-7	`pausableCountGhostSync`	Reachability	Ghost mirror of `pausableCount` is reachable and non-vacuous	✓	✓
CB-8	`globalCountConservation`	Invariant	`sum(pausableCount) == pausables.length` (filtered; see CB-8a, CB-8b)	✓	✓
CB-8a	`globalCountConservation_registerPauser`	Integrity	Global count conservation preserved by `registerPauser`	✗	✗
CB-8b	`globalCountConservation_pause`	Integrity	Global count conservation preserved by `pause`	✓	✓
CB-9	`zeroAddressCount`	Invariant	`pausableCount[address(0)] == 0`	✓	✓
CB-10	`addressZeroNotInArray`	Invariant	`address(0)` never in `pausables` array	✓	✓

CircuitBreaker - Access Control

ID	Name	Type	Description	Audit	Mitig.
CB-11	`onlyAdminChangesPauseDuration`	Access Control	`pauseDuration` changes only when `msg.sender == ADMIN`	✓	✓
CB-12	`onlyAdminChangesHeartbeatInterval`	Access Control	`heartbeatInterval` changes only when `msg.sender == ADMIN`	✓	✓
CB-13	`registryStateOnlyAdminOrPause`	Access Control	Registry state changes only via `registerPauser` (ADMIN) or `pause`	✓	✓
CB-14	`heartbeatExpiryThreeWriters`	Access Control	`heartbeatExpiry` changes only via `registerPauser`, `heartbeat`, or `pause`	✓	✓

CircuitBreaker - Bounds Enforcement

ID	Name	Type	Description	Audit	Mitig.
CB-15	`pauseDurationBounded`	Invariant	`MIN_PAUSE_DURATION <= pauseDuration <= MAX_PAUSE_DURATION` always	✓	✓
CB-16	`heartbeatIntervalBounded`	Invariant	`MIN_HEARTBEAT_INTERVAL <= heartbeatInterval <= MAX_HEARTBEAT_INTERVAL` always	✓	✓

CircuitBreaker - Heartbeat Liveness

ID	Name	Type	Description	Audit	Mitig.
CB-17	`expiredCannotHeartbeat`	Revert Condition	`heartbeat` reverts if `block.timestamp >= heartbeatExpiry[msg.sender]`	✓	✓
CB-18	`expiredCannotPause`	Revert Condition	`pause` reverts if `block.timestamp >= heartbeatExpiry[msg.sender]`	✓	✓
CB-19	`onlyAdminRevivesExpiredPauser`	Parametric	If `heartbeatExpiry` changed and caller is not ADMIN, pauser was live before call	✓	✓
CB-20a	`heartbeatPostCondition_heartbeat`	Integrity	After `heartbeat`: `heartbeatExpiry[sender] == block.timestamp + heartbeatInterval`	✓	✓
CB-20b	`heartbeatPostCondition_pause`	Integrity	After `pause`: `heartbeatExpiry[sender] == block.timestamp + heartbeatInterval`	✓	✓
CB-20c	`heartbeatPostCondition_registerPauser`	Integrity	After `registerPauser(_, nonzero)`: `heartbeatExpiry[pauser] == block.timestamp + heartbeatInterval`	✓	✓

CircuitBreaker - Single-Use Pause

ID	Name	Type	Description	Audit	Mitig.
CB-21	`pauseConsumesRegistration`	Integrity	After `pause(_pausable)`: `pauser[_pausable] == address(0)`	✓	✓
CB-22	`pauseDecrementsCount`	Integrity	After `pause(_pausable)`: `pausableCount[msg.sender]` decreased by 1	✓	✓

CircuitBreaker - Revert Conditions

ID	Name	Type	Description	Audit	Mitig.
CB-23a	`setPauseDurationRevertsNonAdmin`	Revert Condition	`setPauseDuration` reverts if `msg.sender != ADMIN`	✓	✓
CB-23b	`setHeartbeatIntervalRevertsNonAdmin`	Revert Condition	`setHeartbeatInterval` reverts if `msg.sender != ADMIN`	✓	✓
CB-23c	`registerPauserRevertsNonAdmin`	Revert Condition	`registerPauser` reverts if `msg.sender != ADMIN`	✓	✓
CB-24	`heartbeatRevertsUnregistered`	Revert Condition	`heartbeat` reverts if `pausableCount[msg.sender] == 0`	✓	✓
CB-25	`pauseRevertsUnauthorized`	Revert Condition	`pause(_pausable)` reverts if `msg.sender != pauser[_pausable]`	✓	✓
CB-26	`registerPauserRevertsZeroPausable`	Revert Condition	`registerPauser(address(0), _)` reverts	✓	✓

CircuitBreaker - Edge Cases

ID	Name	Type	Description	Audit	Mitig.
CB-27	`idempotentUnregistration`	Integrity	`registerPauser(p, 0)` when `pauser[p] == 0` leaves all state unchanged	✓	✓
CB-28	`replaceWithSelfConservation`	Integrity	`registerPauser(p, q)` when `pauser[p] == q` preserves `pausableCount[q]`	✓	✓

CircuitBreaker - Liveness

ID	Name	Type	Description	Audit	Mitig.
CB-29	`pauseReachable`	Reachability	A registered, live pauser CAN successfully call `pause`	✓	✓
CB-30	`heartbeatReachable`	Reachability	A registered, live pauser CAN successfully call `heartbeat`	✓	✓

Assumptions - Safe

Environment constraints applied to all rules via the `setup` helper function (spec L57-L61):

- `e.msg.value == 0` — all functions are non-payable
- `e.msg.sender != address(0)` — standard EVM constraint (no zero-address caller)
- `e.msg.sender != currentContract` — the contract does not call itself

Assumptions - Proved

The following invariants are used as preconditions via `requireInvariant` in other rules:

- `pauseDurationBounded` (CB-15) and `heartbeatIntervalBounded` (CB-16) — required in nearly all preserved blocks and rules to constrain the initial state within valid bounds
- `forwardIndexConsistency` (CB-1), `reverseIndexConsistency` (CB-2), `membershipEquivalence` (CB-3), `arrayEntriesHavePauser` (CB-4), `addressZeroNotInArray` (CB-10), `cleanStateAfterRemoval` (CB-6), `zeroAddressCount` (CB-9) — cross-referenced in each other's preserved blocks to support inductive reasoning about the enumerable set data structure

Assumptions - Unsafe

- **IPausable NONDET summarization** (spec L43-L44): External calls to `pauseFor(uint256)` and `isPaused()` on registered pausable contracts are summarized as non-deterministic. This means the prover does not model the actual behavior of pausable contracts — properties about the effect of `pause` on the external contract (e.g., "after `pause`, the target is actually paused") are not verified. This is necessary because the pausable implementations are out of scope
- **`optimistic_fallback`**: Enabled in the conf file, allowing the prover to assume external calls to unknown contracts return successfully. Without this, all rules involving `CircuitBreaker::pause` would trivially pass due to the external call reverting

Harness and Helper Contracts

`CircuitBreakerHarness` (harness/CircuitBreakerHarness.sol) inherits `CircuitBreaker` and adds three public getters for internal `Registry.Storage` fields that are not accessible through `CircuitBreaker`'s public interface:

- `getOneBasedIndex(address _pausable)` — returns `registry.oneBasedIndex[_pausable]`, needed for INV-1 through INV-6 (index synchronization and membership equivalence)
- `getPausableAt(uint256 _index)` — returns `registry.pausables[_index]`, needed for INV-1, INV-4, INV-5 (forward index, array entry, and uniqueness checks)
- `getPausablesLength()` — returns `registry.pausables.length`, needed for INV-1, INV-4, INV-8 (array bounds and global count conservation)

The harness passes all constructor parameters through to `CircuitBreaker` unchanged. No production source code was modified.

Verification Results

Final prover run (Certora Prover v8.8.1). 38 of 41 properties verified; 3 known limitations on `registerPauser` inductive step for enumerable set structural invariants.

Spec	Result	Audit Prover URL	Mitig. Prover URL
`CircuitBreaker`	38 verified, 3 known limitations	Prover Link	Prover Link

Known Limitations

Three compensating rules fail for `registerPauser` only: `membershipEquivalence_registerPauser` (CB-3a), `cleanStateAfterRemoval_registerPauser` (CB-6a), `globalCountConservation_registerPauser` (CB-8a).

Root cause: These three structural invariants form a circular inductive dependency. Certora proves invariants inductively: assume the property holds before a function call, execute the function symbolically, prove it still holds after. When multiple invariants depend on each other, a `preserved` block can `requireInvariant` to assume sibling invariants hold at the start. However, invariant A needs invariant B in its preserved block, and B needs A — neither can be proven first.

For example, to prove `membershipEquivalence` (pauser[p] != 0 <=> oneBasedIndex[p] != 0) is preserved by `registerPauser`, the prover needs `cleanStateAfterRemoval` (oneBasedIndex[p] == 0 => pauser[p] == 0) to hold at the start — otherwise it can construct a counterexample starting from an invalid state where some address `p` has `oneBasedIndex[p] == 0` but `pauser[p] != 0`. But proving `cleanStateAfterRemoval` itself requires `membershipEquivalence` to hold at the start. The prover cannot prove both simultaneously.

The `registerPauser` function is uniquely affected because `Registry::setPauser` has 4 distinct code paths (register, replace, unregister, idempotent unregister) that each modify different combinations of the `pauser` mapping, `oneBasedIndex` mapping, `pausables` array, and `pausableCount` mapping. The swap-and-pop removal logic additionally modifies the last array element's index. This path complexity combined with the circular dependency exceeds the prover's ability to close the inductive step. `pause` succeeds because it only exercises the unregister path (1 of the 4 paths).

What was tried: Three automated debug iterations (enriching preserved blocks with all sibling invariants, filtered invariants with compensating per-function rules, and swap-and-pop last-index reasoning patterns) plus two manual experiments (merged invariant combining `membershipEquivalence` and `cleanStateAfterRemoval` into a single property to eliminate the cycle, and explicit `rule` with manual pre-state setup via `require` statements to bypass inductive proof entirely). None resolved the issue — even the manual rule approach failed, indicating the prover's difficulty is with the 4-path symbolic execution of `setPauser` itself, not just the inductive framework.

Coverage: The base invariants are verified for all functions except `registerPauser`. The compensating rules verify for `pause` (which also modifies Registry state via the unregister path). The properties are correct by manual analysis and verified across all other code paths.

Setup and Execution

The Certora Prover can be run either remotely (using Certora's cloud infrastructure) or locally (building from source); both modes share the same initial setup steps.

Common Setup (Steps 1-4)

The instructions below are for Ubuntu 24.04. For step-by-step installation details refer to this setup tutorial.

1. Install Java (tested with JDK 21)

```
sudo apt update
sudo apt install default-jre
java -version
```

2. Install pipx — installs Python CLI tools in isolated environments, avoiding dependency conflicts

```
sudo apt install pipx
pipx ensurepath
```

3. Install Certora CLI

```
pipx install certora-cli==8.8.1
```

4. Install solc-select and the Solidity compiler version required by the project

```
pipx install solc-select
solc-select install 0.8.34
solc-select use 0.8.34
```

Remote Execution

5. Set up Certora key. You can get a free key through the Certora Discord or on their website. Once you have it, export it:

```
echo "export CERTORAKEY=<your_certora_api_key>" >> ~/.bashrc
source ~/.bashrc
```

> **Note:** If a local prover is installed (see below), it takes priority. To force remote execution, add the `--server production` flag:
>
> ```
> certoraRun certora/conf/CircuitBreaker.conf --server production
> ```

Local Execution

Follow the full build instructions in the CertoraProver repository (v8.8.1). Once the local prover is installed it takes priority over the remote cloud by default. Tested on Ubuntu 24.04.

1. Install prerequisites

```
# JDK 19+
sudo apt install openjdk-21-jdk

# SMT solvers (z3 and cvc5 are required, others are optional)
# Download binaries and place them in PATH:
#   z3:   https://github.com/Z3Prover/z3/releases
#   cvc5: https://github.com/cvc5/cvc5/releases
```

```
# LLVM tools
sudo apt install llvm

# Rust 1.81.0+
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
cargo install rustfilt

# Graphviz (optional, for visual reports)
sudo apt install graphviz
```

2. Set up build output directory

```
export CERTORA="$HOME/CertoraProver/target/installed/"
mkdir -p "$CERTORA"
export PATH="$CERTORA:$PATH"
```

3. Clone and build

```
git clone --recurse-submodules https://github.com/Certora/CertoraProver.git
cd CertoraProver
git checkout tags/8.8.1
./gradlew assemble
```

4. Verify installation with test example

```
certoraRun.py -h
cd Public/TestEVM/Counter
certoraRun counter.conf
```

Running Verification

Compilation check (local, fast):

```
certoraRun certora/conf/CircuitBreaker.conf --compilation_steps_only
```

Full prover submission (cloud, blocks until results ready):

```
certoraRun certora/conf/CircuitBreaker.conf --wait_for_results all
```

Run a single rule (for debugging):

```
certoraRun certora/conf/CircuitBreaker.conf --rule forwardIndexConsistency
```

Resources

To learn more about Certora formal verification:

- Updraft Assembly & Formal Verification Course — Comprehensive video course covering assembly and formal verification from the ground up
- Find Highs Using Certora Formal Verification — Practical guide with a companion repo containing simplified examples based on real code and bugs from private audits
- RareSkills Certora Book — Structured tutorial covering CVL syntax, patterns, and common pitfalls
- Alex FV Resources — Curated collection of formal verification resources, examples, and references
- Certora Tutorials — Official Certora documentation and guided tutorials