Formal Verification

Jupiter Lend

January - March 2026

Prepared for Jupiter Lend/Fluid

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Fluid Solana	/Instadapp/fluid-contracts-solana	b4927b6	Solana

Project Overview

This document describes the specification and verification of Jupiter Lend protocol on Solana using Certora prover. The work was undertaken from January 26th 2026 to March 30th 2026.

The main focus for formal verification was on the following programs:

- `Liquidity`
- `Vaults`
- `Lending`

The Certora Prover demonstrated that the implementation of the Solana contracts above is correct with respect to the formal rules designed and written by the Certora team.

The Certora team additionally performed a manual audit that covered the `FlashLoan`, `Oracle` and `LendingRewardRateModel` programs as these were not covered by the formal verification rules. The program `MerkleTree` was not part of the scope. We include the findings of the partial manual review below.

Protocol Overview

Fluid is a capital-efficient DeFi protocol originally built in the EVM ecosystem. Its architecture unifies lending and vaults into a common liquidity layer, enabling high capital reuse, dynamic risk control, and modular composability. Fluid has extended its core design to Solana through a partnership with Jupiter, launching Jupiter Lend (powered by Fluid).

Core Architecture & Design Principles

The core of the protocol is the **Liquidity program.** It tracks total deposits and withdrawals per asset, handles interest accrual, and provides the base liquidity for borrowing and lending.

The liquidity program provides the foundation for multiple sub-protocols, including:

- **Lending program:** Users supply assets to earn yield. Deposits are tracked as fTokens representing a claim on the pooled liquidity. Interest accrual is handled at the module level, leveraging the shared liquidity for efficiency.
- **Vaults program:** Allows over-collateralized borrowing. Collateral and debt are mapped into ticks, discrete risk bags aggregating many user positions. Updates to interest, utilization, and collateral health are performed per tick rather than per individual position. Liquidations also operate at the tick level: when a tick's aggregate health falls below TVL thresholds, the entire tick is liquidated and the effects are distributed proportionally across all positions in that tick. This approach reduces computation and storage overhead and enables batch liquidation of large numbers of positions.
- **Flashloan program:** Provides permissionless, atomic flashloans of liquidity-layer assets. Users can borrow any amount available in the liquidity pool within a single transaction, provided the borrowed amount plus fees is returned before transaction completion.

certora

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	4	4	4
Low	3	3	3
Informational	8	8	2
Total	15	15	9

Severity Matrix

Impact					
	Critical	Medium	Medium/High	High	Critical
	High	Low/Medium	Medium	Medium/High	High
	Medium	Low	Low/Medium	Medium	Medium/High
	Low	Informational	Low	Low/Medium	Medium
		Rare	Unlikely	Likely	Very Likely
			Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
M-01	Token-2022 Extension Validation Is Incomplete	Medium	Fixed
M-02	Liquidation hard-fails when Liquidity Layer withdrawal limits are reached	Medium	Fixed
M-03	Griefing DOS: Repeated Deposit/Withdraw Cycles Suppress Withdrawal Capacity	Medium	Fixed
M-04	Unit Mismatch in Supply/Borrow Ratio Amplification Causes Solvency Drift	Medium	Fixed
L-01	Post-Liquidation debt can be >0 but <Min_debt, breaking some operate() calls	Low	Fixed
L-02	Max Deposit consistently reverts due to +1 lamport rounding charge	Low	Fixed
L-03	Liquidate grants more collateral to liquidator than necessary	Low	Fixed
I-01	Chainlink Staleness Check Uses Optimistic Slot Time Estimate	Informational	Acknowledged
I-02	Redundant max check in get_total_borrow_checked and get_total_supply_checked	Informational	Acknowledged

I-03	Unreachable close_position instruction permanently locks position rent	Informational	Acknowledged
I-04	Incorrect comments in UserBorrowPosition.base_debt_ceiling and max_debt_ceiling	Informational	Acknowledged
I-05	Noop calls in update_user_supply_config and update_user_borrow_config	Informational	Acknowledged
I-06	Redundant max_borrow_limit check in calc_borrow_limit_after_operate()	Informational	Acknowledged
I-07	Use of Magic Numbers for Position Status Instead of Named Enum Variants	Informational	Fixed
I-08	Out-of-bounds access in u256 div_loop function.	Informational	Fixed

Medium Severity Issues

M-01 Token-2022 Extension Validations Are Incomplete

Severity: **Medium**	Impact: **High**	Likelihood: **Unlikely**
Files: crates/library/src/token/spl.rs	Status: Fixed	

Description:
validate_token_extensions_2022() whitelists nine Token-2022 extensions but only validates three of them, and even those three omit authority checks. The result is that a token can pass validation at listing time while retaining live authorities that can later change the extension's behavior. The issues fall into two categories:

1) PermanentDelegate is whitelisted with no validation. A permanent delegate has unconditional, irrevocable authority to transfer or burn tokens from *any* token account of that mint. If a token with an untrusted delegate is listed, the delegate can drain the vault via a standard transfer call. This extension was added to support USDG (a Paxos stablecoin), but there should be validation that the delegate set is the authorised delegate from USDG.
2) Extensions validated on current values but not on the authorities that can change them. TransferFeeConfig, TransferHook, and DefaultAccountState each check a field's current value but never verify that the authority capable of mutating it is None:

Extension	Value checked	Authority checked
TransferFeeConfig	fee == 0	transfer_fee_config_authority == None — no
TransferHook	program_id == None	authority == None — no
DefaultAccountState	state == Initialized	freeze_authority == None — no
MintCloseAuthority	*(nothing)*	close_authority == None — no
PermanentDelegate	*(nothing)*	delegate trusted — no

Every check is a point-in-time snapshot. A token can pass all validation at listing time and have its properties changed afterward:

- A live transfer_fee_config_authority can raise fees up to 100 %; the withdraw_withheld_authority can then harvest them.
- A live TransferHook authority can point transfers at a malicious hook program that blocks vault operations or runs arbitrary logic on every deposit/withdrawal.
- A live freeze_authority can change the default account state to Frozen and directly freeze the vault account.

With 25+ tokens already listed and more planned, relying solely on admin diligence to verify every authority on every extension of every Token-2022 mint is fragile.

Recommendations: For each extension, add a check that the relevant authority is None. If a specific regulated token requires a live authority, allowlist that token's mint or that specific authority pubkey but do not leave the check open for all tokens.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

M-02 Liquidation hard-fails when Liquidity Layer withdrawal limits are reached

Severity: **Medium**	Impact: **High**	Likelihood: **Unlikely**
Files: vaults/src/module/user.rs	Status: Fixed	

Description:

Liquidations perform synchronous CPI withdrawals from the Liquidity Layer to pay liquidators. If the Liquidity Layer's withdrawal limits are reached, the CPI call reverts the entire liquidation transaction. While a withdraw_gap parameter exists to buffer liquidity specifically for liquidations, extreme market outflow scenarios could still deplete this gap, temporarily blocking critical solvency maintenance.

Recommendations:
Implement a robust reserved liquidation buffer that is strictly enforced, or configure the Liquidity Layer to safely bypass standard withdrawal limits during liquidation operations.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

M-03 Griefing DOS: Repeated Deposit/Withdraw Cycles Suppress Withdrawal Capacity		
Severity: **Medium**	Impact: **High**	Likelihood: **Unlikely**
Files: vaults/src/module/user.rs	Status: Fixed	

Description:

An attacker can perform a denial-of-service attack on withdrawals by repeatedly depositing and immediately withdrawing. This exploit manipulates the liquidity protocol's withdrawal limit mechanism to block legitimate users from withdrawing their funds.

The attacker recovers all capital and pays only transaction fees. Repeating the attack resets the decay timer each time, sustaining the freeze indefinitely.

The vault implements a withdrawal gap mechanism to preserve liquidity for liquidations, so this DOS will mainly impact withdrawals, though it can also reduce the liquidity available for liquidations to the gap limit.

Recommendations:
Adjust withdrawal limits on deposits, or enforce a minimum hold time, preventing deposit and withdrawal in the same block.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

M-04 Unit Mismatch in Supply/Borrow Ratio Amplification Causes Solvency Drift

Severity: **Medium**	Impact: **High**	Likelihood: **Unlikely**
Files: Liquidity token_reserve.rs	Status: Fixed	Violated Rule: liquidity_price_update

Description:

calculate_exchange_prices() computes amplification factors that scale the borrow rate into the effective supply rate, accounting for interest-free participants. These factors are derived from get_with_interest_vs_free_ratio(), which divides total_supply_with_interest by total_supply_interest_free (and likewise for borrows).

The two values use incompatible units. The with interest fields store raw tokens (divided by the exchange price at deposit time), while interest free fields store face-value tokens. At genesis the exchange price is 1.0, so the units coincide and the ratio is correct. As exchange prices grow, raw tokens become progressively smaller than the value they represent, and the ratio diverges.

The same unit mismatch affects the branch conditions that select which amplification formula to apply (self.total_supply_with_interest < self.total_supply_interest_free and likewise for borrows). The two branches use structurally different formulas, where one inverts the ratio, so selecting the wrong branch compounds the error beyond the ratio drift alone.

Example: With a 2.0x exchange price, a pool containing 1,000 face-value tokens of with-interest supply (stored as 500 raw) and 500 interest-free tokens should have an amplification of 1.5x (1,500 / 1,000). The code sees a 50/50 raw split and computes 2.0x instead — overpaying with-interest suppliers by ~33%.

Recommendation: Convert raw tokens to face value before computing the ratio: multiply with interest by the current exchange price and divide by EXCHANGE_PRICES_PRECISION.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

Low Severity Issues

L-01 Post-Liquidation debt can be >0 but <Min_debt, breaking some operate() calls		
Severity: **Low**	Impact: **Low**	Likelihood: **Unlikely**
Files: vaults/src/state/structs.rs	Status: Fixed	

Description:
A partially liquidated position can end up with non-zero debt below the protocol's `MIN_DEBT` floor, creating a dead zone where most `operate` calls revert.

In `fetch_latest_position`, a position survives liquidation if its remaining debt exceeds 1% of the original:

```rust
// structs.rs:125
if position_raw_debt > initial_position_raw_debt.safe_div(100)? {
    position_raw_debt = position_raw_debt
        .safe_mul(FOUR_DECIMALS - 1)?
        .safe_div(FOUR_DECIMALS)?;
} else {
    position_raw_debt = 0;
}
```

However, `add_debt_to_tick` unconditionally enforces an absolute minimum on any non-zero debt being re-entered into the tick system:

```rust
// operate.rs:194
if net_debt_raw < min_debt {
    return Err(error!(ErrorCodes::VaultUserDebtTooLow));
}
```

When a position's post-liquidation debt passes the 1% survival check but falls below `MIN_DEBT`, the position becomes partially inoperable. Any `operate` call that keeps debt non-zero — collateral deposit, partial payback, or additional borrow — reverts at the `add_debt_to_tick` check. Only a full payback (`new_debt = i128::MIN`), which sets `debt_raw` to zero and bypasses `add_debt_to_tick` entirely, succeeds.

This requires the original debt to be below `100 × MIN_DEBT` (approximately 100 USDC or 0.0001 SOL), so only dust-scale positions are affected.

Recommendations:
In `fetch_latest_position`, zero out surviving positions that fall below `MIN_DEBT`

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

L-02 Max Deposit consistently reverts due to +1 lamport rounding charge		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: vaults/src/module/user.rs	Status: Fixed	

Description:

The protocol applies a `safe_add(1)` increment to the user's unscaled deposit amount to enforce safety rounding. If a user attempts a "Max Deposit" of their exact wallet balance, the transfer attempts to pull `balance + 1`, causing an insufficient funds revert.

Recommendations:
Prevent adding the 1 lamport rounding charge if it would exceed the user's available balance, or deduct it internally from the credited position amount.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

L-03 Liquidate grants more collateral to liquidator than necessary		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: vaults/src/module/user.rs	Status: Fixed	Violated Rule: verify_summary_ execute_liquidation

Description:

The protocol computes the debt to be liquidated and corresponding collateral as below:

```rust
//user.rs:769

// Calculate debt to liquidate
let mut debt_liquidated: u128 = current_data.get_debt_liquidated(col_per_debt)?;

// Calculate collateral to liquidate
// extremely unlikely to overflow considering debt_liquidated is realistically within
u64.
let mut col_liquidated: u128 = debt_liquidated
      .safe_mul(col_per_debt)?
      .safe_div(10u128.pow(RATE_OUTPUT_DECIMALS))?;

// Adjust for edge case
if current_data.debt == debt_liquidated {
      debt_liquidated = debt_liquidated.safe_sub(1)?;
}
```

In the edge case, the `debt_liquidated` is reduced, but `col_liquidated` is not adjusted accordingly. This results in the liquidator being paid `col_per_debt` units of collateral more than necessary. The solvency of the protocol is affected by this, although the impact is quite low in absolute numbers.

Recommendations:
Compute `col_liquidated` after `debt_liquidated` is updated for the edge case.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

Informational Severity Issues

I-01 Chainlink Staleness Check Uses Optimistic Slot Time Estimate

Description:
The Chainlink staleness check converts a slot delta to elapsed wall-clock time using a hardcoded AVG_SLOT_TIME_IN_MILLISECONDS of 400ms:

```rust
let time_elapsed = current_slot
    .safe_sub(publish_slot)?
    .safe_mul(AVG_SLOT_TIME_IN_MILLISECONDS)?
    .safe_div(1000)?;
```

Since Solana slot times routinely exceed 400ms during network congestion (often reaching 500–800ms), this calculation underestimates the real wall-clock age of a Chainlink price. A price that is actually 15 minutes old could be estimated as only 10 minutes old and pass the MAX_AGE_OPERATE (600s) threshold.

This is the only oracle source affected — Pyth and Redstone use verify_publish_time(), which compares unix timestamps directly. Chainlink v2 on Solana does not expose a unix timestamp in its feed data, so slot-based estimation is the only available approach.

The current behavior may be a deliberate liveness tradeoff: a more conservative (higher) slot time estimate would reject prices more aggressively during congestion, which is precisely when fresh prices are hardest to obtain. Using the optimistic 400ms value keeps the protocol operational during degraded network conditions rather than reverting all operate and liquidate calls.

Recommendations:
If the leniency during congestion is deliberate, add a code comment explaining the tradeoff. If strict wall-clock freshness is desired, increase `AVG_SLOT_TIME_IN_MILLISECONDS` to a more conservative value (e.g., 500–600ms), accepting the risk of reverts during congestion.

Customer's response: Acknowledged (accepted tradeoff; `AVG_SLOT_TIME_IN_MILLISECONDS` left at 400ms unless stricter freshness is required)

I-02 Redundant max check in get_total_borrow_checked and get_total_supply_checked

Description:
In `get_total_borrow_checked`:

```rust
if amount > 0
    && (normalized_interest_borrow > MAX_TOKEN_AMOUNT_CAP
        || total_borrow > MAX_TOKEN_AMOUNT_CAP)
```

And identically in `get_total_supply_checked`:

```rust
if amount > 0
    && (normalized_interest_supply > MAX_TOKEN_AMOUNT_CAP
        || total_supply > MAX_TOKEN_AMOUNT_CAP)
```

Since `total = normalized_interest + interest_free` and `interest_free >= 0`, `normalized > CAP` always implies `total > CAP`. The `normalized > CAP` condition is a strict subset of `total > CAP` and can never be true when `total > CAP` is false. The check is therefore redundant.

Recommendations:
Simplify both checks to:

```rust
if amount > 0 && total_borrow > MAX_TOKEN_AMOUNT_CAP
if amount > 0 && total_supply > MAX_TOKEN_AMOUNT_CAP
```

Customer's response: Acknowledged (comment-only fix deferred)

I-03 Unreachable close_position instruction permanently locks position rent

Description:
The `close_position()` function is implemented in `vaults/src/module/user.rs` internally but omitted from the public program dispatcher in `lib.rs`. Consequently, users cannot execute the instruction to close empty positions and reclaim their Solana rent, resulting in minor but permanent state bloat.

Recommendations:
Expose the close_position instruction in the lib.rs library entry point. or remove the dead code

Customer's response: Acknowledged

I-04 Incorrect comments in UserBorrowPosition.base_debt_ceiling and max_debt_ceiling

Description:
The UserBorrowPosition struct has incorrect comments describing the `base_debt_ceiling` and `max_debt_ceiling` fields:

```rust
pub base_debt_ceiling: u64, // base debt ceiling: below this, there's no debt ceiling limits (normal or raw depends on with_interest)

pub max_debt_ceiling: u64, // base debt ceiling: below this, there's no debt ceiling limits (normal or raw depends on with_interest)
```

For both it states that below the ceiling, no debt ceiling limits apply, which does not match the implemented functionality.
For `base_debt_ceiling`: if the limit is below `base_debt_ceiling` then `base_debt_ceiling` is used as the limit.
For `max_debt_ceiling`: if the limit is **above** `max_debt_ceiling` , then `max_debt_ceiling` is used as the limit.

Recommendations:
Update the comments to match the implementation

Customer's response: Acknowledged (comment-only fix deferred)

I-05 Noop calls in update_user_supply_config and update_user_borrow_config

Description:
In the liquidity admin module's `update_user_borrow_config()` and `update_user_supply_config()` function, there are calls to `set_interest_mode(mode)` in the branch where `mode == with_interest`.
The calls in admin.rs:511 and admin.rs:659 will set the value to the current value, effectively doing nothing.

Recommendations:
The call to set_interest_mode() can be removed in these branches.

Customer's response: Acknowledged (redundant `set_interest_mode`() removal deferred)

I-06 Redundant max_borrow_limit check in calc_borrow_limit_after_operate()

Description:
`calc_borrow_limit_after_operate()` shrinks the `borrow_limit` if the post-operate limit is less than the pre-operate limit. When the post-operate limit exceeds the pre-operate limit, it returns the pre-operate limit.

Since the pre-operate limit will always be `<= max_borrow_limit`, the following check is redundant:

```rust
if borrow_limit > max_borrow_limit {
    borrow_limit = max_borrow_limit;
}
```

If `borrow_limit > max_borrow_limit`, it will automatically be `> new_borrow_limit` and be caught by the next condition:

```rust
if new_borrow_limit > borrow_limit {
    return Ok(borrow_limit);
}
```

Recommendations:
Consider removing the redundant max_borrow_limit check.

Customer's response: Acknowledged (redundant guard kept for safety)

I-07 Use of Magic Numbers for Position Status Instead of Named Enum Variants

Description:
In the liquidity admin module, the `pause_user()` and `unpause_user()` functions assign raw integer literals (0 and 1) directly to the status field of user supply and borrow positions, rather than using the named enum variants (e.g., `UserSupplyPositionStatus::Active`, `UserSupplyPositionStatus::Paused`, and their borrow counterparts). This reduces code clarity and maintainability: a reader must cross-reference the enum definition to understand the intent, and future additions of new status values increase the risk of silent mismatches if raw literals are scattered across the codebase.
Using the typed enum variants — or the existing convenience methods such as `set_status_as_active()` — would make the intent self-documenting and reduce the chance of introducing incorrect status values through copy-paste or refactoring errors.

Recommendations:
Consider the following changes:

- `user_supply_position.status = 1` replace by `user_supply_position.status = UserSupplyPositionStatus::Paused as u8`
- `user_supply_position.status = 0` replace by `user_supply_position.status = UserSupplyPositionStatus::Active as u8`

or
`user_supply_position.set_status_as_active()`

- `user_borrow_position.status = 1` replace by
  `user_borrow_position.status = UserBorrowPositionStatus::Paused as u8`
- `user_borrow_position.status = 0` replace by
  `user_borrow_position.status = UserBorrowPositionStatus::Active as u8`
  Or
  `user_borrow_position.set_status_as_active()`
- `if supply_status.unwrap() != 1` replace by
  `if supply_status.unwrap() != UserSupplyPositionStatus::Paused as u8`
- `if borrow_status.unwrap() != 1` replace by
  `if borrow_status.unwrap() != UserBorrowPositionStatus::Paused as u8`

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

I-08 Out-of-bounds access in u256 div_loop function.

Description:
File u256.rs contains an out-of-bounds array access bug in the `div_loop` function (line 644). When dividing a 4-word dividend by a 3- or 4-word divisor, the Knuth Algorithm D "add-back" correction step reads items[4] on a 4-element array, causing a panic.
The public API functions `safe_multiply_divide`, `safe_multiply_divide_result`, and `safe_multiply_divide_up` all construct the divisor as `U256Muldiv::new(0, c)` where c is a u128. This produces at most a 2-word divisor, for which the correction step is provably unreachable.
Recommendations:
Read from *`dividend_carry_space` when `use_carry` is true, mirroring lines 619–630.

Customer's response: Acknowledged - fixed in PR #168

Fix Review: Fixed

certora

Formal Verification

Verification Methodology

We performed verification of the **Fluid/JupiterLend** protocol 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 (CVLR) and the implementation source code written in Rust.
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 CVLR 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/initialization functionality 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 initialization function. 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. Each such case results in one rule.

In the following, we provide the verification plan for each contract in their individual section. We provide the specification, the assumptions and finally, the verification results.

certora

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.

General Assumptions

- Configuration:
  - Solana Platform Tools - v1.43.
  - Loops are unrolled at most once.
- In all formally verified protocols, the SafeMath library has been substituted to perform the same arithmetic as before, but via Certora Prover NativeInts Library instead of u64/u128/u256. Certora's NativeInts Library is already formally verified, and translates directly to the underlying SMT reasoning. This reduces the processing time for verification tasks.

Vaults Contract

Verification Plan

The formal verification of the Vaults protocol focused on proving that the protocol preserves solvency. This translates to the following three main tasks:

1. Verifying that `operate` and `liquidate` preserve solvency. In particular, `operate` must consider both cases: (i) the position is stale and must be updated via `fetch_latest_position`, or (ii) the position is updated (in other words, latest). For `liquidate`, the verification must consider liquidation of at least 2 ticks to cover all paths.
2. Verification of `fetch_latest_position` function.
3. Verification of `absorb` function.

Let us consider these tasks in detail.

Vaults Solvency Formulation:

To define solvency of the Vaults contract, we define the following terms:

1. `TotalSupply`: `vault_state.total_supply`
2. `TotalBorrow`: `vault_state.total_borrow`
3. `LiquiditySupply`: `vault_user_supply_position.amount`
4. `LiquidityBorrow`: `vault_user_borrow_position.amount`
5. `CurrentUserSupply`: `supply of an up-to-date position`
6. `SumCurrentUserSupply`: `sum of supply of an up-to-date position for all positions`
7. `CurrentUserDebt`: `debt of an up-to-date position`
8. `SumCurrentUserDebt`: `sum of debt of an up-to-date position for all positions`
9. `BranchDebt`: `amount of debt stored in a branch`
10. `TickDebt`: `amount of debt stored in a tick`
11. `SumBranchDebt`: `sum of amount of debt stored in a branch over all branches`
12. `SumTickDebt`: `sum of amount of debt stored in a tick over all ticks`
13. `VaultSupplyPrice`: `vault_state.vault_supply_ex_price`
14. `VaultBorrowPrice`: `vault_state.vault_borrow_ex_price`
15. `LiquiditySupplyPrice`: `supply_token_reserve.supply_exchange_price`
16. `LiquidityBorrowPrice`: `borrow_token_reserve.borrow_exchange_price`
17. `OracleSupplyPrice`: `oracle supply price`

18. `OracleBorrowPrice`: oracle borrow price

The following properties define solvency of the vaults contract:

(P-A) Relating vault aggregate with liquidity:

- `VaultSupplyPrice * TotalSupply <= LiquiditySupplyPrice * LiquiditySupply`
- `VaultBorrowPrice * TotalBorrow >= LiquidityBorrowPrice * LiquidityBorrow`

(P-B) Relating vault aggregate with user positions:

- `TotalSupply <= SumCurrentUserSupply`
- `TotalBorrow <= SumCurrentUserDebt`

(P-C) Relating vault aggregate with branches and ticks:

```
TotalBorrow == SumBranchDebt + SumTickDebt
```

(P-D) Relating branches with user positions:

```
BranchDebt == sum_{all positions that point to that branch} CurrentUserDebt

TickDebt == sum_{all positions that point to that tick} CurrentUserDebt
```

(P-E) Vault Solvency:

```
VaultSupplyPrice * TotalSupply * OracleSupplyPrice >=

    VaultBorrowPrice * TotalBorrow * OracleBorrowPrice
```

We prove properties (P-A) to (P-E) for functions `operate` and `liquidate`.

Verification setup and general assumptions

- For `operate`, we use a summary for fetch_latest_position function. We later verify this summary with properties for fetch_latest_position.
- For `liquidate`, we refactor the liquidation loop into a separate function `execute_liquidation` to make it verification-friendly. We use a summary of `execute_liquidation` when verifying `liquidate`. This summary is verified with rule `verify_summary_execute_liquidation`.
- We use two branches with fixed ids 2 and 1, with the `vault_state.current_branch_id` == 2. Branch 2 is connected to branch 1.
- We use a summary for `get_ticks_from_racle_prices`. This summary is verified with rule `verify_summary_get_ticks_from_oracle_price`.
- In function `get_actual_amounts`, we disable the path when `actual_debt_amt` > `debt_amout`. This path was reviewed manually.
- The call to liquidity `operate` from vaults is mocked to modify only the `user_supply_position` and `user_borrow_position`, as the property (P-A) only cares about the position of vaults program at liquidity.
- We use a summary of the `get_new_position_info` function. This summary is verified by rule `verify_get_new_position_info_lemma`.
- Both borrow and supply tokens are assumed to be 9 decimals. Thus, `scale` and `unscale` functions do not change the input.
- Function `check_if_position_safe` is assumed to be nondeterministic, except for rule `operate_position_safe_tick`.
- For all rules involving a user position except `operate_position_safe_tick`, the initial tick is assumed to be 1 and final tick is assumed to be between 0 and 2. Thus, a position's tick can move in either direction, allowing us to catch bugs in either direction.

- For rule `operate_position_safe_tick`, collateral factor ratio is fixed at 2 with `oracle_exchange_price` (of supply in terms of borrow) at 2 and vault supply and borrow prices at 1. The corresponding collateral factor tick is 462.
- Liquidity prices are assumed to be 1 (in 1e12), vault prices are assumed to be nondeterministically bigger than 1 (in 1e12). This allows us to simplify computation in liquidity, while allowing us to catch rounding errors in vaults.
- Following functions are assumed to return nondeterministic values: `mint_position_token_and_remove_authority`, `check_if_withdrawal_safe_for_withdrawal_gap`, `check_if_ratio_safe_for_deposit_or_payback`, and `get_debt_liquidated`.

certora

Program Properties

(P-01) Correctness of Vaults operate: (P-A).

Status: Verified

Rule Name	Status	Description	Links
liquidity_solvency_operate _fetch_deposit_and_borrow	Verified	*Check (P-A) for operate case: position is stale, deposit and borrow.*	*Rule report* *Rule report*
liquidity_solvency_operate _fetch_deposit_and_payback	Verified	*Check (P-A) for operate case: position is stale, deposit and payback.*	
liquidity_solvency_operate _fetch_withdraw_and_borrow	Verified	*Check (P-A) for operate case: position is stale, withdraw and borrow.*	
liquidity_solvency_operate _fetch_withdraw_and_payback	Verified	*Check (P-A) for operate case: position is stale, withdraw and payback.*	
liquidity_solvency_operate _latest_deposit_and_borrow	Verified	*Check (P-A) for operate case: position is latest, deposit and borrow.*	
liquidity_solvency_operate _latest_deposit_and_payback	Verified	*Check (P-A) for operate case: position is latest, deposit and payback.*	
liquidity_solvency_operate _latest_withdraw_and_borrow	Verified	*Check (P-A) for operate case: position is latest, withdraw and borrow.*	
liquidity_solvency_operate _latest_withdraw_and_payback	Verified	*Check (P-A) for operate case: position is latest, withdraw and payback.*	

certora

(P-02) Correctness of Vaults operate: (P-B, P-C, P-D, P-E).

Status: Verified

Rule Name	Status	Description	Links
integrity_position _operate_fetch	Verified	*Check (P-B) for operate case: position is stale*	*Rule report* *Rule report*
integrity_position _operate_latest	Verified	*Check (P-B) for operate case: position is latest*	
integrity_tick _operate_fetch	Verified	*Check (P-C) and (P-D) for operate: position is stale*	
integrity_tick _operate_latest	Verified	*Check (P-C) and (P-D) for operate: position is latest*	
integrity_vault_totals _operate_fetch **integrity_vault_totals _operate_latest** **operate_position_safe_tick** **safe_tick_implies_position _solvency** **vault_solvency_lemma**	Verified	*Check (P-E) for operate. The proof is decomposed into following steps:* *(i) operate_vault_totals: ensures vault total borrows and supplies are updated accordingly (split for cases position is latest and position is stale) .* *(ii) operate_position_safe_tick: operate ensures the position's tick is in safe zone (below collateral factor).* *(iii) safe_tick_implies_position_solvency: if a position's tick is in safe zone, then it is solvent.* *(iv) vault_solvency_lemma: if each position is solvent, then the vault is solvent.*	

certora

(P-03) Correctness of Vaults liquidate.

Status: Verified after fix

Rule Name	Status	Description	Links
liquidity_solvency_liquidate	Verified	*Check (P-A) for liquidate.*	*Rule report* *Rule report* *Rule report*
integrity_debt_execute_liquidation	Verified	*Check (P-C) for liquidate.*	
integrity_debt_factor_execute_liquidation	Verified	*Check (P-D) for liquidate.*	
Vaults_solvency_liquidate **verify_summary_execute_liquidation**	Verified after fix	*Check (P-E) for liquidate. The proof is relies on the summary for execute_liquidation that failed earlier due to L-03.*	
integrity_slippage_liquidate	Verified	*Check that liquidation respects slippage*	

(P-04) Verifying summaries

Status: Verified

Rule Name	Status	Description	Links
verify_get_new_position_info_lemma	Verified	*Verifying summary for get_new_position_info.*	*Rule report*
verify_summary_get_ticks_from_oracle_price	Verified	*Verifying summary for get_ticks_from_oracle_price.*	

Verification of fetch_latest_position function:

The following properties must hold for the `fetch_latest_position` function:
(P-A) If a position is fully liquidated, then output debt and collateral of the position must be zero.
(P-B) If the output debt is zero, then collateral must also be zero.
(P-C) The debt of a stale position must reduce.
(P-D) When debt is nonzero, the output debt and collateral must be related by the ratio determined by the new tick of the position.

These properties are necessary to prove the correctness of the operate function.

Verification setup and general assumptions

- We use 2 branches, with id 2 and 1. The position's liquidation branch is 2, which is connected to 1. Branch 2 is assumed to be `Merged` and branch 1 is assumed to be `Liquidated`. This covers all code paths of the `fetch_latest_position` function.
- We assume that all ticks are positive. We reset `COLD_TICK` to 0 and `MIN_TICK` to 1 for this purpose.
- We mock the function `get_tick_status` to always retrieve data from index 0 in `tick_id_liquidation`.

Program Properties

(P-05) Correctness of the fetch_latest_position function.

Status: Verified

Rule Name	Status	Description	Links
integrity_fully_liquidated_fetch	Verified	*Check (P-A) for fetch_latest_position.*	*Rule report*
integrity_debt_zero_implies_col_zero_fetch	Verified	*Check (P-B) for fetch_latest_position.*	
integrity_debt_is_reduced_fetch	Verified	*Check (P-C) for fetch_latest_position.*	
integrity_relates_debt_and_col_fetch	Verified	*Check (P-D) for fetch_latest_position.*	

Verification of absorb function:

For `absorb`, the following properties are critical to the functioning of the protocol:
(P-A) The function `absorb` must not update the `total_supply`, `total_borrow` as well exchange prices stored in the `vault_state`. It should also not change the minima ticks associated with branches. Finally, it should ensure that `vault_state.topmost_tick <= max_tick` after `absorb` is run.
(P-B) The function `absorb` must update the following fields of `vault_state` correctly: `topmost_tick`, `is_branch_liquidated`, `current_branch_id` and `total_branch_id`.
(P-C) The function `absorb` must update the `connected_branch_id` and `connected_minima_tick` for absorbed branches correctly. The branch status must also be updated to reflect the post-state.
(P-D) The output `absorbed_col` and `absorbed_debt` must be computed correctly.

For absorb, we assume the setup of 2 branches with id 2 and 1, with branch 2 connected to branch 1. We split the verification into the following six cases to cover all code paths:
(Case 1): `vault_state.is_branch_liquidated() == false && branch1.minima_tick == COLD_TICK.`
(Case 2): `vault_state.is_branch_liquidated() == true`
    (Subcase 2a): `branch2.minima_tick <= max_tick && branch2.minima_tick <= next_tick.`
    (Subcase 2b): `branch2.minima_tick <= max_tick && branch2.minima_tick > next_tick.`
    (Subcase 2c): `branch2.minima_tick > max_tick && branch1.minima_tick == COLD_TICK.`
    (Subcase 2d): `branch2.minima_tick > max_tick && branch1.minima_tick > COLD_TICK && branch1.minima_tick <= next_tick.`
    (Subcase 2e): `branch2.minima_tick > max_tick && branch1.minima_tick > COLD_TICK && branch1.minima_tick > next_tick && branch1.minima_tick < max_tick.`

We prove properties (P-B, P-C, P-D) for all six cases. Property P-A can be proved without the case split.

Verification setup and general assumptions

- We use two branches with fixed ids 2 and 1, with the `vault_state.current_branch_id == 2`. Branch 2 is connected to branch 1.
- We assume that all ticks are positive. We reset `COLD_TICK` to 0 and `MIN_TICK` to 1 for this purpose.
- We mock the function `fetch_next_tick_absorb` to return `next_tick <= max_tick` and constant `absorbed_col` and `absorbed_debt`. These constants are then used to evaluate the post-state.

certora

Program Properties

(P-06) Function absorb updates vault_state correctly: (P-A, P-B)

Status: Verified

Rule Name	Status	Description	Links
integrity_basic_absorb	Verified	*Check (P-A) for absorb*	*Rule report*
integrity_vault_state_absorb_case1	Verified	*Check (P-B) for absorb: case 1*	
integrity_vault_state_absorb_case2a	Verified	*Check (P-B) for absorb: case 2a*	
integrity_vault_state_absorb_case2b	Verified	*Check (P-B) for absorb: case 2b*	
integrity_vault_state_absorb_case2c	Verified	*Check (P-B) for absorb: case 2c*	
integrity_vault_state_absorb_case2d	Verified	*Check (P-B) for absorb: case 2d*	
integrity_vault_state_absorb_case2e	Verified	*Check (P-B) for absorb: case 2e*	

certora

(P-07) Function absorb updates branches correctly: (P-C)

Status: Verified

Rule Name	Status	Description	Links
integrity_branch_absorb_case1	Verified	*Check (P-C) for absorb: case 1*	*Rule report*
integrity_branch_absorb_case2a	Verified	*Check (P-C) for absorb: case 2a*	
integrity_branch_absorb_case2b	Verified	*Check (P-C) for absorb: case 2b*	
integrity_branch_absorb_case2c	Verified	*Check (P-C) for absorb: case 2c*	
integrity_branch_absorb_case2d	Verified	*Check (P-C) for absorb: case 2d*	
integrity_branch_absorb_case2e	Verified	*Check (P-C) for absorb: case 2e*	

certora

(P-08) Function absorb computes absorbed col and debt correctly: (P-D)

Status: Verified

Rule Name	Status	Description	Links
integrity_col_debt_absorb_case1	Verified	*Check (P-D) for absorb: case 1*	*Rule report*
integrity_col_debt_absorb_case2a	Verified	*Check (P-D) for absorb: case 2a*	
integrity_col_debt_absorb_case2b	Verified	*Check (P-D) for absorb: case 2b*	
integrity_col_debt_absorb_case2c	Verified	*Check (P-D) for absorb: case 2c*	
integrity_col_debt_absorb_case2d	Verified	*Check (P-D) for absorb: case 2d*	
integrity_col_debt_absorb_case2e	Verified	*Check (P-D) for absorb: case 2e*	

certora

Verification of u256 library:

For the u256 library, we show that `safe_multiply_divide` and `safe_multiply_divide_up` compute the correct result when the input parameters are all within the u64 range.

Program Properties

(P-09) Verifying safe_multiply_divide and safe_multiply_divide_up

Status: Verified

Rule Name	Status	Description	Links
safe_multiply_divide_integrity	Verified	*Verifying safe_multiply_divide.*	*Rule report*
safe_multiply_divide_up_integrity	Verified	*Verifying safe_multiply_divide_up*	

Liquidity Contract

Verification Plan

The formal verification of the Liquidity contract focuses around proving solvency of the token reserve. We state the solvency as an invariant of the Liquidity contract. We begin by defining the following terms:

```
TokenSupply_wi: token_reserve.total_supply_with_interest

TokenSupply_if: token_reserve.total_supply_interest_free

TokenBorrow_wi: token_reserve.total_borrow_with_interest

TokenBorrow_if: token_reserve.total_borrow_interest_free

SupplyPrice: token_reserve.supply_exchange_price

BorrowPrice: token_reserve.borrow_exchange_price

TokenTotalClaim: token_reserve.total_claim_amount

TokenTotalSupply: TokenSupply_if * EXCHANGE_PRICE_PRECISION
                    + TokenSupply_wi * SupplyPrice

TokenTotalBorrow: TokenBorrow_if * EXCHANGE_PRICE_PRECISION
                    + TokenBorrow_wi * BorrowPrice

VaultBalance: balanceOf(TokenReserve.vault)
```

With the above terms defined, we can define liquidity solvency invariant as follows:

```
TokenTotalSupply + TokenTotalClaim <= VaultBalance + TokenTotalBorrow
```

The LHS represents the amount (in actual units) that the token reserve owes to its suppliers. The RHS represents the amount that the token reserve currently has plus the amount expected from the borrowers. The solvency of the token reserve simply states that the token reserve must expect to receive more than it owes to its suppliers.

We prove the solvency invariant by establishing that all functions of the Liquidity contract preserve it. Since admin functions do not change the above terms, we focus on `operate` and `update_exchange_price`. Particularly for `operate`, we check for deposit, withdraw, borrow and payback code paths individually. This forms the first property:

(P-A) Solvency Invariant is preserved by `operate`, `update_exchange_price, claim` and `collect_revenue`.

In order to establish solvency invariant, the following property must also be established:

(P-B) The token reserve stores `last_utilization` correctly.

This is also proven as an invariant over `operate` and `update_exchange_price`. We call this the "Last Utilization Invariant". In addition to solvency invariants, following integrity properties are crucial to correct functioning of the liquidity contract:

(P-C) `TokenSupply_wi` is the sum of all `user_supply_positions.amount` for all user supply positions with interest. Similar is true for `TokenSupply_if`.

(P-D) `TokenBorrow_wi` is the sum of all `user_borrow_positions.amount` for all user borrow positions with interest. Similar is true for `TokenBorrow_if`.

(P-E) `TokenTotalClaim` amount must be the sum of all `user_claim.amount`.

We prove properties (P-C, P-D, P-E) over `operate`.

General Assumptions

- When proving properties (P-A, P-B, P-C, P-D, P-E) over `operate`, we assume that the prices are already updated. This is justified because the correctness of price update logic is checked when proving solvency invariant for `update_exchange_price`.
- When proving the solvency invariant for `update_exchange_price,` we assume that `total_supply_interest_free` and `total_borrow_interest_free` are zero. This is justified because only the Flashloan contract is set without interest. Note that, Flashloan contract updates prices before it is active, and does not allow price updates once it is active. Additionally, the flashloan must be paid off at the end of the transaction. Hence, the interest free quantities are set to zero at the end.

- The parameters of `operate` are `i64` instead of `i128`.
- The RateModel function `calc_borrow_rate_from_utilization` is mocked to return nondeterministic values.

Program Properties

(P-10) Liquidity Solvency Invariant

Status: Verified after fix

Rule Name	Status	Description	Links
liquidity_solvency _operate_deposit	Verified	*Case: operate_deposit*	*Rule report*
liquidity_solvency _operate_withdraw	Verified	*Case: operate_withdraw*	
liquidity_solvency _operate_borrow	Verified	*Case: operate_borrow*	
liquidity_solvency _operate_payback	Verified	*Case: operate_payback*	
liquidity_price_update _only_with_interest **liquidity_price_update** **update_price_fee _period_lemma**	Verified after fix	*Case: update_exchange_price. Note that the rule liquidity_price_update failed earlier due to M-04.* *The proof is decomposed into following steps:* *(i) liquidity_price_update_only_with_interest: proving solvency holds for price_update with non-zero values of interest_free supply and borrow quantities. Fixes some values like last_update_timestamp to make verification feasible.* *(ii) liquidity_price_update: proving solvency holds for price_update with interest_free values assumed to be zero.* *(iii) update_price_fee_period_lemma: A lemma necessary to prove liquidity_price_update.*	
liquidity_solvency _claim	Verified	*Case: claim*	
liquidity_solvency	Verified	*Case: collect_revenue.*	

_collect_revenue			

(P-11) Last Utilization Invariant

Status: Verified

Rule Name	Status	Description	Links
liquidity_operate_deposit_utilization	Verified	*Case: operate_deposit.*	*Rule report*
liquidity_operate_withdraw_utilization	Verified	*Case: operate_withdraw.*	
liquidity_operate_borrow_utilization	Verified	*Case: operate_borrow.*	
liquidity_operate_payback_utilization	Verified	*Case: operate_payback.*	
liquidity_price_update_utilization	Verified	*Case: update_exchange_price.*	

certora

(P-12) Operate Integrity Properties

Status: Verified

Rule Name	Status	Description	Links
operate_integrity _deposit_or_withdraw	Verified	*Function operate satisfies property P-3a in case of deposit or withdraw.*	*Rule report*
operate_integrity _borrow_or_payback	Verified	*Function operate satisfies property P-3b in case of borrow or payback.*	
operate_borrow _claim_integrity	Verified	*Function operate satisfies property P-3c in case of borrow.*	
operate_withdraw _claim_integrity	Verified	*Function operate satisfies property P-3c in case of withdraw.*	

Lending Contract

Verification Plan

The formal verification of the Lending contract focuses on proving its solvency. This involves comparing the obligations that Lending owes to its users (depositors) versus the assets it owns at the liquidity layer. We begin by defining the following terms:

`FTokenSupply`: `lending.f_token_mint.supply`

`FTokenPrice`: `lending.token_exchange_price`

`SupplyPrice` : `token_reserve.supply_exchange_price` (`token_reserve` for lending's underlying asset)

`LendingPositionAmount`: `lending_user_supply_position.amount`

`LendingAssets`: `FTokenSupply * FTokenPrice`

`LiquidityAssets`: `LendingPositionAmount * SupplyPrice`

Note that, `LendingAssets` represents the assets that Lending owes to its users. On the other hand, `LiquidityAssets` represents the assets that belong to the Lending contract at the liquidity layer. Solvency can be stated as:

`LendingAssets <= LiquidityAssets`.

However, due to lending rewards, it becomes clear that solvency in above terms does not hold. In such a situation, it is useful to analyze the difference in assets, i.e.

```
LendingAssets - LiquidityAssets
```

We would like to determine the upper bound of how much the delta between the two assets can grow with one call to any Lending function. If before a function call,

```
LendingAssets - LiquidityAssets <= k
```

holds for some k, then after the function call

```
LendingAssets' - LiquidityAssets' <= k + Bound
```

must hold. Here, `LendingAssets'` and `LiquidityAssets'` represent the value of those terms after the function call. The Certora team determined this bound and mathematically proved it with the Certora prover. The bound is as follows:

Bound = `FTokenSupply * (FTokenPrice' - FTokenPrice)`

Intuitively, the Lending contract loses value equal to the additional value it offers to existing `FToken` holders due to increase in `FTokenPrice`. Note that this only comes into play if rewards are set for the Lending program. It was manually reviewed that the second component matches the rewards set by the LendingRewardsRateModel. Finally, the property we prove is the following:

(P-4) (Bound on assets delta) `LendingAssets'` - `LiquidityAssets'` <=

`LendingAssets` - `LiquidityAssets` + `Bound`.

We prove this for the functions `deposit`, `mint`, `redeem` and `withdraw`. We ignore the equivalent functions with slippage protection as they call the same internal functions.

General Assumptions

- The liquidity prices stored at the Lending program are already updated. Similarly, the token_reserve prices are already updated before the function call. This is justified because these price updates are analyzed separately.
- The mint, burn and transfer functions are replaced with already formally verified CVLR versions of these functions.
- The call to liquidity operate from lending is mocked to modify only the user_supply_position, as the property (P-4) only cares about the position of Lending program at liquidity.
- `calculate_new_token_exchange_price` has been over-approximated to return a nondeterministic price between 1*EXCHANGE_PRICE_PRECISION and 2*EXCHANGE_PRICE_PRECISION.

certora

Program Properties

(P-13) Bound on assets delta

Status: Verified

Rule Name	Status	Description	Links
lending_solvency_deposit	Verified	*Case: deposit.*	*Rule report*
lending_solvency_mint	Verified	*Case: mint*	
lending_solvency_redeem	Verified	*Case: redeem*	
lending_solvency_withdraw	Verified	*Case: withdraw*	

certora

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.