Security Assessment & Formal Verification Final Report

Kamino Lending

Feb 2025

Prepared for Kamino Finance

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Klend	https://github.com/Kamino-Finance/klend	5af6cfc	Solana

Project Overview

This document describes the specification and verification of **Kamino Lending Protocol** using the **Certora Prover**, as well as **manual code review** findings of two consecutive audits. The work was undertaken from **November 4, 2024** to **December 13, 2024**, as well as from **February 3, 2025** to **February 21, 2025**.

All contracts in the following folder are included in our scope:

`klend/programs/klend/src/*`

The Certora Prover demonstrated that the implementation of the Solana contracts above is **correct with respect to the formal properties** formulated and written by the Certora team. In addition, the team performed a **manual audit** of all the Solana contracts. During the verification process and the manual audit, the Certora team discovered bugs in the protocol code and reviewed their fixes both manually and formally, as listed on the following pages.

Protocol Overview

Kamino Lending (KLend) is a decentralized finance (DeFi) platform designed to optimize the borrowing and lending of crypto assets on the Solana blockchain. Its primary goal is to address inefficiencies in traditional and decentralized lending systems, such as high fees, slow transaction speeds, and lack of transparency, by leveraging Solana's high throughput and low latency.

Core Concept and Architecture:

KLend operates as a non-custodial lending protocol, where users interact directly with smart contracts without intermediaries. The platform enables users to:

- **Lend Assets**: Deposit cryptocurrencies into liquidity pools to earn interest.
- **Borrow Assets**: Use collateralized loans to access liquidity without liquidating their holdings.
- **Flash Loans:** Undercollateralized loans to access liquidity with same block repayment

Key actors in the system include:

- **Lenders**: Provide liquidity to the system and earn yield based on loan interest.
- **Borrowers**: Collateralize their crypto assets to access loans.
- **Liquidators**: Maintain system health by liquidating undercollateralized positions when market conditions fluctuate.

Interfaces and Interactions:

KLend integrates with other DeFi protocols and ecosystem tools to enhance functionality:

- **Oracle Services**: Collaborates with Solana-based oracles to fetch accurate price feeds, ensuring proper collateral valuation.
- **Other DeFi Protocols**: May interact with Solana liquidity protocols or staking systems to enhance user options for asset utilization.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	2	2	2
Low	1	1	-
Informational	7	7	2
Total	10	8	

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
M-01	First depositor attack	Medium	Fixed in 5af6cfc
M-02	Precision loss in exchange rate can allow user to receive more then they have deposited due to rounding error	Medium	Fixed in 5af6cfc
L-01	Flash loan fees discrepancy	Low	Fix by configuration change.
I-01	The rate used in approximate_compounded_interest() does not reflect the yearly rate	Informational	No fix necessary
I-02	Function duplicate	Informational	Fixed
I-03	Utilization_limit not used in withdrawals	Informational	Acknowledged
I-04	Naming mismatch	Informational	Fixed
I-05	Duplicate checks	Informational	Acknowledged
I-06	Unneeded if statement in check_refresh_ixs	Informational	Acknowledged

I-07	reset_elevation_group_debts() does not check if accounts in borrow_reserves_iter matches obligation.borrows	Informational	Acknowledged

Medium Severity Issues

M-01 First depositor attack		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: reserve.rs	Status: Fixed in 5af6cfc	

Description:

A first depositor can inflate the share price, causing future depositors to lose their liquidity due to the rounding of the amount of shares to be minted.

While it is not possible to "externally donate" tokens to the protocol in a way that will manipulate the `exchange_rate`, it is still possible to "donate" tokens to the protocol through the `deposit()` function. In a scenario where there is only 1 share, any new deposit of an amount smaller than the current liquidity will not cause any new shares to be minted.
As the amount of liquidity grows exponentially from one deposit of this type to another, a first depositor can repeat this process and inflate the share price in a significant way with a feasible amount of deposits, causing any future deposit by a different user that is smaller than the current liquidity to be lost to the first depositor, as the number of minted shares would be rounded down to 0.

Exploit Scenario:

```Python
running 1 test

test lending_market::tests_market::test_first_depositor ...

deposit_collateral(....,liquidity_amount: 2, ..)

Deposit:  amount: 2.0000 value: 2.0000

Reserve: available: 2  / borrowed:  0

====================

borrow_obligation_liquidity( ..., borrow: 1, ...)

Requested 1, allowed 1

Last refreshed borrows (outside elevation group) 0

Reserve: available: 1  / collateral supply:  2

Borrow:  amount: 1.0000 value: 1.0000 value_bf: 1.0000

Deposit:  amount: 2.0000 value: 2.0000

====================

** NEXT SLOT **

====================

** current_borrow_rate . utilization_rate = 0.5

Borrow:  amount: 1.0000 value: 1.0000 value_bf: 1.0000

Deposit:  amount: 2.0000 value: 2.0000

repay_obligation_liquidity(..., liquidity_amount: 1, ...)

CalculateRepayResult 1 / 1.0000

calculate_repay

REPAY: 1 / 1
```

```
Last refreshed borrows (outside elevation group) 2

obligation.borrow_factor_adjusted_debt_value_sf = 1.0000

Obligation new borrowed value after repay 0.0000 for

Borrow:  amount: 0.0000 value: 0.0000 value_bf: 0.0000

Deposit:  amount: 2.0000 value: 2.0000

 Exchange rate BPS: 9999

 Reserve: total liquidity: 2.0000  / collateral supply:  2

withdraw_obligation_collateral(..., collateral_amount: 1, ...)

redeem_reserve_collateral(..., collateral_amount: 1, ...)

Borrow:  amount: 0.0000 value: 0.0000 value_bf: 0.0000

Deposit:  amount: 1.0000 value: 1.0000

========== start deposits to inflate exchangerate ==========

 Exchange rate BPS: 9999

 Reserve: total liquidity: 1.000000000554921359  / collateral supply:  1

deposit 1

 Exchange rate BPS: 4999

 Reserve: total liquidity: 2.000000000554921359  / collateral supply:  1

deposit 2

 Exchange rate BPS: 2499

 Reserve: total liquidity: 4.000000000554921359  / collateral supply:  1

deposit 4

 Exchange rate BPS: 1249

 Reserve: total liquidity: 8.000000000554921359  / collateral supply:  1

deposit 8
```

```
 Exchange rate BPS: 624

 Reserve: total liquidity: 16.000000000554921359  / collateral supply:  1

deposit 16

 Exchange rate BPS: 312

 Reserve: total liquidity: 32.000000000554921359  / collateral supply:  1

deposit 32

 Exchange rate BPS: 156

 Reserve: total liquidity: 64.000000000554921359  / collateral supply:  1

deposit 64

 Exchange rate BPS: 78

 Reserve: total liquidity: 128.000000000554921359  / collateral supply:  1

ok
```

Recommendations: Make sure that it is not possible for any user to deposit tokens into an empty pool. This could be achieved, for example, by introducing a virtual user which owns a certain amount of shares that could never be withdrawn.

Customer's response: Fixed in 5af6cfc.

Fix Review: Two changes have been made to prevent this issue from happening.
The first adds an initial deposit on reserve creation. No actual ctokens are minted, which ensures that the initial deposit cannot be withdrawn. The initial deposit limits the impact of rounding errors to manipulate the exchange rate, making the the attack practically impossible.
The second assures that a user is not overpaying for their ctokens. It recalculates the liquidity to pay for minting the ctokens to receive. In case a user would receive less ctokens due to rounding errors or a manipulated exchange rate, the actual liquidity they are charged is calculated from that ctoken amount. In the POC result above that would mean they would be charged 0 liquidity when they receive 0 ctokens.
Each of these changes alone would be sufficient to prevent the attack.

M-02 Precision loss in exchange rate can allow user to receive more then they have deposited due to rounding error

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: lending_operations.rs	Status: Fixed in commit 5af6cfc	Violated Properties: *ratio_increase_redeem_reserve_collateral* *ratio_increase_post_liquidate_redeem* *reserve_solvency_redeem_reserve_collateral* *reserve_solvency_post_liquidate_redeem*

Description:

The exchange rate between collateral and liquidity is calculated by dividing collateral total supply by total liquidity. The result is represented as a Fraction, having 63 bits for the integer part and 60 bits for the fractional part.

```
Unset
/// Return the current collateral exchange rate.
fn exchange_rate(&self, total_liquidity: Fraction) -> LendingResult<CollateralExchangeRate> {
    let rate = if self.mint_total_supply == 0 || total_liquidity == Fraction::ZERO {
        INITIAL_COLLATERAL_RATE
    } else {
        Fraction::from(self.mint_total_supply) / total_liquidity
    };
    Ok(CollateralExchangeRate(rate))
}
```

When calculating the liquidity to return upon redeeming collateral, the collateral amount is divided by this exchange rate:

```
Unset
impl CollateralExchangeRate {
    /// Convert reserve collateral to liquidity
    pub fn fraction_collateral_to_liquidity(&self, collateral_amount: Fraction) -> Fraction {
        collateral_amount / self.0
    }
}
```

As the exchange rate is rounded down and used to calculate redeemable amounts by division, the amount of liquidity to withdraw can round up.
Note that the exchange rate is represented as a Fraction with 60 bits for the fractional parts. This rounding error only occurs if the amount of collateral is larger than 2^59. Currently there do not exist any tokens with significant value on the Solana ecosystem that have this large of a total supply, hence at the moment of this audit an exploit is likely not possible.

Exploit Scenario:

```
Python

Reserve: total liquidity:   15000000

Reserve: collateral supply: 10000000

exchange_rate: 0.666666666666666666


deposit   4600000000000000000 liquidity

received  3066666666666666664 collateral

exchange_rate: 0.666666888888962962

redeem    3066666666666666664 collateral
```

```
received  4600000000000000005 liquidity
Reserve: total liquidity:   14999995

Reserve: collateral supply: 10000000
```

Recommendations: It is recommended to use Mul-Div pattern for this calculation and to ensure rounding and precision loss will always be in favour of the protocol.

Customer's response: Fixed in commit 5af6cfc.

Fix Review: The fix applies the Mul-Div pattern for all exchange rate calculations and is correctly implemented.

Low Severity Issues

L-01 Flash loan fees discrepancy		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: lending_operations.rs	Status: Confirmed	

Description:
If there's a large enough difference between `reserve.config.fees.flash_loan_fee` for two different reserves, then it might be possible for a user to pay less fees for a flash loan than intended.

Say, for example, that there's a 0.3% fee for a flash loan of Token A, and 0% fee for a flash loan of Token B. In that scenario, a user can flash loan a certain amount of Token B and then use it as a collateral to borrow Token A and hold a large amount of it momentarily. That user can then repay the loan in the same block for the Token B they deposited and return the flash loan, paying a smaller fee than they would have if they just flash-loaned Token A directly.

Exploit Scenario:

```Python
kJITOSOL_SOL reserve:  GFLLgX9V6PR1PzhgsGzeyUddpt2ws2PBRAxRwoLrboGp

kJITOSOL_SOL flashloan fee: 0.003

kJITOSOL_SOL borrow fee: 0.0000001

kJITOSOL_SOL LTV percent: 75
```

```
sol  flashloan fee: 0

sol borrow fee: 0

sol  LTV percent: 65

normal flashloan:

=================

flashloaned 1000000 kJITOSOL_SOL, repay including fees:  1003000

Total Fees  3000

loan sol, borrow kJITOSOL_SOL:

==============================

flashloan 1550000  sol

deposit 1550000  sol collateral

Deposit:  amount: 1550000.0000 value: 1550000.0000

borrow 1000000 kJITOSOL_SOL

Repay loan 1000001 kJITOSOL_SOL

Withdraw and redeem sol collateral

redeemed_amount 1550000

repaid flashloan 1550000  sol, repay including fees 1550000

Total Fees  1
```

Recommendations: Have the same flash loan fee for all borrowable tokens.

Customer's response: Acknowledged. Will be fixed by a configuration change.

Informational Severity Issues

I-01. The rate used in approximate_compounded_interest() does not reflect the yearly rate

Description: The function `approximate_compounded_rate()` calculates the growth of debt with time. As interest is being compounded regardless of whether the function is being called or not, the yearly rate parameter could be misleading as it does not represent the actual yearly rate. For example, a yearly rate of 10% would result in an actual yearly rate of $e^{0.1} - 1 \approx 10.5\%$.

Recommendation: Make sure that the borrowers understand that the yearly rate does not include the compounded interest or consider changing the code.

Alternatively, if it is preferred that the yearly rate parameter would represent the actual yearly rate, the code can be changed. A simple change could be to convert the rate into $ln(1 + rate)$ at the beginning of the function. One way to approximate this could be using the Taylor approximation $ln(1 + rate) \approx rate + rate^3/3 - rate^2/2$.

Customer's response: The function represents APY and is correct.

I-02. Two functions have the exact same implementation

Description: The functions `collateral_exchange_rate()` and `collateral_exchange_rate_ceil()` have the same implementation.

Recommendation: Consider using just one of these functions.

Customer's response: Fixed in f3b2ce1.

I-03. Utilization limit only applies to borrows but not to withdrawals

Description: Attempting to borrow a large enough amount of tokens might revert due to the `utilization_rate` exceeding the `utilization_limit` but this limitation does not apply to withdrawals.

```
Unset
if new_utilization_rate >= Fraction::from_percent(utilization_limit)
       && utilization_limit != 0 {
        msg!(
            "Borrowing above utilization rate is disabled, current {}, new {}, limit {}",
            current_utilization.to_display(),
            new_utilization_rate.to_display(),
            utilization_limit
        );
        return err!(LendingError::BorrowingAboveUtilizationRateDisabled);
}
```

A user can overcome this restriction by first depositing a large enough amount of liquidity and then borrowing the amount that they wished to borrow. After that, the large amount of liquidity that was initially deposited by the user could be withdrawn.

Recommendation: Consider deleting this restriction if it is not necessary.

Customer's response: Acknowledged.

I-04. Mismatch function name default_padding_173() and array length 172

Description: In `lending_markets.rs` the function `default_padding_173()` is used as the default value for an array of length 172

```
Unset
#[cfg(feature = "serde")]
fn default_padding_173() -> [u64; 172] {
        [0; 172]
}
```

Recommendation: Change function name to `default_padding_172()`

Customer's response: Fixed.

I-05. duplicate checks in liquidate obligation handler process function

Description: In `handler_liquidate_obligation_and_redeem_reserve_collateral::process()` the following calls are made to check the input:

```
lending_checks::liquidate_obligation_checks()

lending_checks::redeem_reserve_collateral_checks()
```

Most of the checks performed inside the `redeem_reserve_collateral_checks()` are also done in the `liquidate_obligation_checks()` function. The only additional check in `redeem_reserve_collateral_checks()` verifies that `user_destination_liquidity` cannot be the `supply_vault`, which is not needed anymore when called from `redeem_reserve_collateral::process()`.

Recommendation: Move the check `if reserve.liquidity.supply_vault == accounts.user_destination_liquidity.key()` to `liquidate_obligation_checks()` and remove the call to `redeem_reserve_collateral_checks` from the `handler_liquidate_obligation_and_redeem_reserve_collateral::process()` function.

Customer's response: Acknowledged.

I-06. Unnecessary if statement in check_refresh_ixs macro

Description: The following codeblock is from `check_refresh_ixs` in `utils/macro.rs`

```
Unset
($ctx_accounts:expr, $reserve_one:expr, $reserve_two:expr, $mode_one:expr, $mode_two:expr) =>
{{
    let _reserve_one = $reserve_one.load()?;
    let _reserve_two = $reserve_two.load()?;

    if $reserve_one.key() == $reserve_two.key() {
        $crate::utils::check_refresh(
            &$ctx_accounts.instruction_sysvar_account,
            &[
                ($reserve_one.key(), &_reserve_one),
                ($reserve_one.key(), &_reserve_one),
            ],
            &$ctx_accounts.obligation.to_account_info().key(),
            &[$mode_one, $mode_two],
        )?;
    } else {
        $crate::utils::check_refresh(
            &$ctx_accounts.instruction_sysvar_account,
            &[
                ($reserve_one.key(), &_reserve_one),
                ($reserve_two.key(), &_reserve_two),
            ],
            &$ctx_accounts.obligation.to_account_info().key(),
            &[$mode_one, $mode_two],
        )?;
    }
}};
```

An if-statement is used to check if reserve_one and reserve_two are the same and the `check_refresh` function is called with (1) `reserve_one, reserve_one` if both pubkeys are the same or (2) `reserve_one, reserve_two` when the pubkeys are different.
This effectively makes the functionality in the if part the same as in the else part.

Recommendation: Remove the if/else construct and only use the code that is currently in the else.

Customer's response: Acknowledged.

I-07. reset_elevation_group_debts() does not check if accounts in borrow_reserves_iter matches obligation.borrows

Description: When `reset_elevation_group_debts` is called when `elevation_group` is set to None, the reserves in `borrow_reserves_iter` are not checked to be the same as those in obligation.borrows.
This could cause the incorrect reserves to be updated.
Currently `reset_elevation_group_debts` is only called from `request_elevation_group()`, which also calls `refresh_obligation_borrows` where this check is done. Because of this there is no direct risk of this being exploited.

Recommendation: To have this secure on a functional level, it is recommended that an extra check is added to check if the passed in accounts match those of the obligation:

```
Unset
let mut obligation_borrows_iter = obligation
        .borrows
        .iter_mut()
        .filter(|borrow| borrow.borrow_reserve != Pubkey::default());

+        require_keys_eq!(
+            borrow.borrow_reserve,
+            reserve.get_pubkey(),
+            LendingError::InvalidAccountInput
+        );
```

Customer's response: Acknowledged.

Formal Verification

Methodology

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/init() functionality as a base case, and consider all other (relevant) externally callable functions 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.

Note that to make verification more tractable, we sometimes prove on lower level functions that contain the relevant logic. In the case of Kamino, we prove invariants correct by proving properties on the relevant functionality provided in `lending_operations.rs`.

General Assumptions and Simplifications

Configuration and Munging

1) For tractability, we chose an **approximation of the `Fraction` datatype**. This datatype is used to represent fixed point numbers in Solana and is a wrapper for Rust library `fixed` using `FixedU128<60>`. The code is based on `Fraction128<60>` where 68 bits represent the integer part of the number and 60 are the fractional bits. We use fractional numbers of type `FixedU64<14>` with 50 bits used for the integer and 14 bits used for the fractional part. This makes the verification task simpler on the verification engine, while still allowing us to catch potential rounding errors.

2) Loops are inherently difficult for formal verification. We handle loops by unrolling them a specific number of times. We thus use an **underapproximation on the quantity of**

borrows and deposits in `Obligation` positions. In our setting, an obligation position contains at most two different deposits and one borrow. Consequently, we use a **loop_iter of 2**, unrolling each loop exactly two times.

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 counterexample exists that violates one of the assertions of the property.

Formal Verification Properties

Account Health

<u>Module General Assumptions</u>

These assumptions apply to all rules in the Account Health category. All of these assumptions are for tool performance reasons and to make verification tractable:

- The functions `update_elevation_group_debt_trackers_on_borrow`, `update_elevation_group_debt_trackers_on_repay`, and `post_deposit_obligation_invariants` are replaced with empty functions for verification.
- The function `calculate_liquidation` is replaced with a function that generates a `Result<CalculateLiquidationResult>` with arbitrary (nondeterministic) values. This is an overapproximation of the original code. A verified rule means the prover covers all possible return values from this function.
- We assume `get_deposit_asset_tiers` and `get_borrows_asset_tiers` return empty vectors.
- Vectors of accounts (or serialization objects holding account_infos) given as parameters to these lending operations are restricted to size 2

Module Properties

P-01. A healthy account cannot become unhealthy by user operation	
Status: Verified	Specification: If the LTV of an obligation is below the unhealthy LTV ratio before performing an arbitrary user operation, and if we assume prices do not change and interest is not accrued, the LTV remains below the unhealthy LTV after the user operation.

Rule Name	Status	Description	Link to rule report
rule_health_status_init_obligation	Verified	*Base Case: init_obligation()* *Note that for this property we consider an account to be "healthy" if it has a deposit value of 0; however, note that in P-2 we prove an invariant that accounts with deposit value 0 must also have a borrow value of 0.*	*Report Part 1*
rule_health_status_borrow_obligation_liquidity	Verified	*Step Case: borrow_obligation_liquidity()*	
rule_health_status_deposit_obligation_collateral	*Verified*	*Step Case: deposit_obligation_collateral()*	
rule_health_status_refresh_obligation_borrow	Verified	*Step Case: refresh_obligation_borrows()*	

s			
rule_health_status_refresh_obligation_deposits	Verified	*Step Case: refresh_obligation_deposits()*	
rule_health_status_repay_obligation_liquidity	Verified	*Step Case: repay_obligation_liquidity()*	
rule_health_status_socialize_loss	Verified	*Step Case: socialize_loss()*	
rule_health_status_withdraw_obligation_collateral	Verified	*Step Case: withdraw_obligation_collateral()*	
rule_health_status_liquidate_obligation	Verified	*Step Case: liquidate_obligation()*	[Report Part 2]()
rule_health_status_request_elevation_group	Verified	*Step Case: request_elevation_group()* *For tool performance reasons and to make verification tractable here we assume there are no referrers (referrer_token_states_iter is empty).*	

P-02. An account with no collateral must have no borrows	
Status: Verified	Specification: We prove an invariant that if an obligation has zero deposited value, it must also have zero borrowed value (we show that both the borrow_factor_adjusted_debt and borrowed_assets_market are both zero when deposit value is zero). Rather than using implication we specify this with the logical equivalent: `!(obligation.deposited == 0) || (obligation.borrow == 0)`

Rule Name	Status	Description	Link to rule report
rule_no_col_no_debt_init_obligation	Verified	*Base Case: init_obligation()*	[Report Part 1]()
rule_no_col_no_debt_borrow_obligation_liquidity	Verified	*Step Case: borrow_obligation_liquidity()*	
rule_no_col_no_debt_deposit_obligation_collateral	Verified	*Step Case: deposit_obligation_collateral()*	
rule_no_col_no_debt_refresh_obligation_borrows	Verified	*Step Case: refresh_obligation_borrows()*	

rule_no_col_no_debt_refresh_obligation_deposits	Verified	*Step Case: refresh_obligation_deposits()*	
rule_no_col_no_debt_repay_obligation_liquidity	Verified	*Step Case: repay_obligation_liquidity()*	
rule_no_col_no_debt_socialize_loss	Verified	*Step Case: socialize_loss()*	
rule_no_col_no_debt_withdraw_obligation_collateral	Verified	*Step Case: withdraw_obligation_collateral()*	
rule_no_col_no_debt_liquidate_obligation	Verified	*Step Case: liquidate_obligation()*	[Report Part 2]
rule_no_col_no_debt_request_elevation_group	Verified	*Step Case: request_elevation_group()* *For tool performance reasons and to make verification tractable here we assume there are no referrers (referrer_token_states_iter is empty).*	

P-03. A healthy user cannot lose collateral	
Status: Verified	Specification: We prove that if an account (obligation) is healthy (using the same definition as P-01) initially, the collateral of the account before an arbitrary lending operation is greater or equal to the collateral after calling that operation. An exception to this is withdraw_collateral for which this property does not hold, thus we cannot prove it.

Rule Name	Status	Description	Link to rule report
rule_no_col_decrease_borrow_obligation_liquidity	Verified	*Rule: No collateral decrease on borrow_obligation_liquidity()*	[Report Part 1]
rule_no_col_decrease_deposit_obligation_collateral	Verified	*Rule: No collateral decrease on deposit_obligation_collateral()*	
rule_no_col_decrease_refresh_obligation_borrows	Verified	*Rule: No collateral decrease on refresh_obligation_borrows()*	
rule_no_col_decrease_refresh_obligation_deposits	Verified	*Rule: No collateral decrease on refresh_obligation_deposits()*	

rule_no_col_decrease_repay_obligation_liquidity	Verified	*Rule: No collateral decrease on repay_obligation_liquidity()*	
rule_no_col_decrease_socialize_loss	Verified	*Rule: No collateral decrease on socialize_loss()*	
rule_no_col_decrease_liquidate_obligation	Verified	*Rule: No collateral decrease on liquidate_obligation()*	[*Report Part 2*]
rule_no_col_decrease_request_elevation_group	Verified	*Rule: No collateral decrease on request_elevation_group* *For tool performance reasons and to make verification tractable here we assume there are no referrers (referrer_token_states_iter is empty).*	

Protocol Solvency

Module General Assumptions

These assumptions apply to all rules in the Protocol Solvency category. All of these assumptions are for tool performance reasons and to make verification tractable:

- The functions `post_deposit_obligation_invariants` and `validate_obligation_asset_tiers` are replaced with empty functions for verification.
- The function `get_liquidation_params` is replaced with summary that returns nondeterministic `LiquidationParams` with additional assumptions that `user_ltv` + `liquidation_bonus_rate` is less than `Fraction::ONE,` and the `liquidation_bonus_rate` being non-zero.
- The function `accrue_interest` is replaced with a summary that behaves the same as the original function, however uses `BigFraction` (rather than `U256`) which is mapped to `Fraction` in the Certora verification setting.
- The function `approximate_compounded_interest` is abstracted with a nondeterministic return value assumed to be `≥ Fraction::ONE`.
- The function `current_borrow_rate` is replaced with a function returning a nondeterministic `Fraction ≤ Fraction::ONE`, that is 0 ≤ borrow_rate ≤ 1.
- `BorrowRateCurve`: The function `get_borrow_rate` is replaced with a nondeterministic return value of type Fraction. The function `validate` is replaced with an empty summary.
- Vectors of accounts (or serialization objects holding account_infos) given as parameters to these lending operations are restricted to size 2.
- The functionality for flash loans was not part of the verification effort but has been manually audited. Any found issues are reported above.

Module Properties

P-04. Vault Solvency: sum of deposited/withdrawn liquidity covers collateral shares of a reserve	
Status: Fully verified post fix	Specification: This property specifies that no lending operation can make a liquid reserve illiquid by means of accidental withdraws, rounding errors or other issues, that is the **total sum of deposits (and withdraws) ≥ amount of minted shares**. Additional Assumptions/Munging: Since the reserve accounting did not provide a field that simply tracks deposited liquidity regardless of fees, borrows and interest accrual, we introduce a new field in LiquidityReserve: `u64 sum_of_deposits.` The field is updated accordingly on deposits to and withdraws from a liquidity reserve in order to prove that the accounting over all external functions is correct. We further prove interest accrual separately, and assume a reserve to be fresh for further step cases to alleviate the prover.

Rule Name	Status	Description	Link to rule report
vault_solvency_init_reserve	Verified	*Base Case: init_reserve()*	*Full Report*
vault_solvency_interest_accrual	Verified	*Step Case: interest accrual* *Note that this rule is proved separately to safely assume interest has already been accrued in the following rules covering further step cases. The full proof is thus compositional in order to be tractable for the underlying verification engine.*	

vault_solvency_refresh_reserve	Verified	*Step Case: refresh_reserve()*	
vault_solvency_borrow_obligation_liquidity	Verified	*Step Case: borrow_obligation_liquidity()*	

deposit_reserve_liquidity	Verified	*Step Case: deposit_reserve_liquidity()*
		
		Requires an additional assumption proved in rule reserve_solvency_deposit_reserve_liquidity to circumvent spurious counterexamples.
		`reserve.liquidity.total_supply()`
		`>= reserve.collateral.mint_supply`	

deposit_obligation_collateral	Verified	*Step Case: deposit_obligation_collateral()*	
vault_solvency_liquidate_obligation_repay	Verified	*Step Case: liquidate_obligation* *Proved on argument repay_reserve* *For simplicity we assume no elevation_group for the obligation argument.*	
vault_solvency_liquidate_obligation_withdraw	Verified	*Step Case: liquidate_obligation* *Proved on argument withdraw_reserve* *For simplicity we assume no elevation_group for the obligation argument.*	
vault_solvency	Verified	*Step Case: redeem_fees()*	

_redeem_fees			
vault_solvency_socialize_loss	Verified	*Step Case: socialize_loss()* *Note that this property is verified on socialize_loss() as socialize_loss() can only decrease the borrowed amount of a reserve, but does not directly affect the sum of deposits/withdraws.*	
vault_solvency_update_reserve_config	Verified	*Step Case: update_reserve_config()* *Note that this case requires a specific prover option* `-solanaSkipCallRegInst true`	
vault_solvency_withdraw_obligation_collateral	Verified	*Step Case: withdraw_obligation_collateral()*	
vault_solvency_withdraw_referrer_fees	Verified	*Step Case: withdraw_referrer_fees()*	
vault_solvency_redeem_reserve_collateral	Verified post fix	*Step Case: redeem_reserve_collateral()*	
vault_solvency_post_liquidate_redeem	Verified post fix	*Step Case: post_liquidate_redeem()*	

P-05. Reserve Solvency: sum of all assets of a reserve cover its collateral shares	
Status: Fully verified post fix	Specification: This property specifies that a reserve's total assets given by reserve.liquidity.total_supply() should cover at least the amount of minted shares, i.e. **reserve.liquidity.total_supply() ≥ reserve.collateral.mint.supply**. This form of solvency covers borrows, interest accrual and fees as well as deposited liquidity. We prove by induction that no liquid reserve can be illiquid by means of a user operation. An exception to this property is `socialize_loss` for which this property does not hold, thus we cannot prove it. However, this is a privileged operation only to be invoked in case of a bad debt position.

Rule Name	Status	Description	Link to rule report
reserve_solvency_init_reserve	Verified	*Base Case: init_reserve()*	[Report Part 1]()
reserve_solvency_interest_accrual	Verified	*Step Case: interest accrual* *Note that this rule is proved separately to safely assume interest has already accrued in the following rules covering further step cases. The full proof is thus compositional in order to be tractable for the underlying verification engine.*	
reserve_solvency_refresh_reserve	Verified	*Step Case: refresh_reserve()*	

reserve_solvency_borrow_obligation_liquidity	Verified	*Step Case: borrow_obligation_liquidity()*	
reserve_solvency_deposit_reserve_liquidity	Verified	*Step Case: deposit_reserve_liquidity()*	
reserve_solvency_deposit_obligation_collateral	Verified	*Step Case: deposit_obligation_collateral()*	
reserve_solvency_liquidate_obligation_repay	Verified	*Step Case: liquidate_obligation()* *Proved on argument repay_reserve* *For simplicity we assume no elevation_group for the obligation argument.*	
reserve_solvency_liquidate_obligation_withdraw	Verified	*Step Case: liquidate_obligation()* *Proved on argument withdraw_reserve* *For simplicity we assume no elevation_group for the obligation argument.*	
reserve_solvency_redeem_fees	Verified	*Step Case: redeem_fees()*	
reserve_solvency_update_reserve_config	Verified	*Step Case: update_reserve_config()* *Note that this case requires a specific prover option* `-solanaSkipCallRegInst true`	

reserve_solvency_withdraw_obligation_collateral	Verified	*Step Case: withdraw_obligation_collateral()*	
reserve_solvency_withdraw_referrer_fees	Verified	*Step Case: withdraw_referrer_fees()*	
reserve_solvency_redeem_reserve_collateral	Verified post fix	*Step Case: redeem_reserve_collateral()* *This property was violated due to a precision loss in the calculation of the collateral_exchange_rate as described in* *M-02*.	*Report Part 2*
reserve_solvency_post_liquidate_redeem	Verified post fix	*Step Case: post_liquidate_redeem()* *This property was violated due to a precision loss in the calculation of the collateral_exchange_rate as described in* *M-02*.	*Report Part 3*

P-06. Supply/Collateral ratio only decreases on liquidation of bad debt (socialize_loss)

Status: Verified post fix	Specification: This property verifies that the value of collateral shares of a reserve can only increase upon interest accrual as well as user actions, that is collateral shares/total assets increases. The only way to decrease the value of collateral is by liquidating bad debt, i.e. `socialize_loss()`.

Rule Name	Status	Description	Link to rule report
ratio_increase_on_interest_accrual	Verified	*accrue_interest() increases the value of collateral shares.*	[Report Part 1]
ratio_increase_borrow_obligation_liquidity	Verified	*borrow_obligation_liquidity() does not decrease the value of collateral shares.*	
ratio_increase_refresh_reserve	Verified	*refresh_reserve() does not decrease the value of collateral shares.*	
ratio_increase_deposit_reserve_liquidity	Verified	*deposit_reserve_liquidity() does not decrease the value of collateral shares.*	

ratio_increase_deposit_obligation_collateral	Verified	*deposit_obligation_collateral() does not decrease the value of collateral shares.*	
ratio_increase_redeem_fees	Verified	*redeem_fees() does not decrease the value of collateral_shares.*	
ratio_increase_update_reserve_config	Verified	*update_reserve_config() does not decrease the value of collateral shares.* *Note that this case requires a specific prover option* `-solanaSkipCallRegInst true`	
ratio_increase_withdraw_obligation_collateral	Verified	*withdraw_obligation_collateral() does not decrease the value of collateral shares.*	
ratio_increase_redeem_reserve_collateral_case1	Verified post fix	*redeem_reserve_collater() does not decrease the value of collateral shares.* *This property was violated due to [M-02]*	
ratio_increase_redeem_reserve_collateral_case2	Verified post fix	*redeem_reserve_collater() does not decrease the value of collateral shares.* *This property was violated due to [M-02]*	
ratio_increase_post_liquidate_redeem_case1	Verified post fix	*post_liquidate_redeem() does not decrease the value of collateral shares.* *This property was violated due to [M-02]*	

ratio_increase_post_liquidate_redeem_case2	Verified post fix	*post_liquidate_redeem() does not decrease the value of collateral shares.* *This property was violated due to [M-02]*	
ratio_decrease_socialize_loss	Verified	*socialize_loss() decreases the value of collateral shares.*	
ratio_increase_liquidate_obligation_withdraw	Verified	*liquidate_obligation() does not decrease the value of collateral shares of the withdraw reserve.*	*Report Part 2*
ratio_increase_liquidate_obligation_repay	Verified	*liquidate_obligation() does not decrease the value of collateral shares of the repay reserve.*	*Report Part 3*

Liquidations

Module General Assumptions

These assumptions apply to all rules in this section. All of these assumptions are for tool performance reasons and to make verification tractable:

- All rules in this section deal with a single obligation. This obligation has a single borrow liquidity and single collateral, both stored at index 0. For further simplification, we assume the price of 1 for both borrow and collateral, so that the obligation's market values and borrow/collateral amounts match. This results into following assumptions (with appropriate type conversion) :

```
Unset
liquidity.market_value_sf == liquidity.borrowed_amount_sf

collateral.market_value_sf == collateral.deposited_amount

obligation.borrow_factor_adjusted_debt_value_sf == liquidity.borrowed_amount_sf

obligation.deposited_value_sf = collateral.deposited_amount
```

- We further assume that the obligation belongs to no elevation groups and the lending market does not allow auto-deleveraging.
- To reduce the number of nonlinear operations in a function, we summarize certain functions. The summarized functions are conditionally compiled using Cargo features. Below is an overview of the functions that we have summarized and the cargo feature under which they are in scope:
    - `get_liquidation_params_summary`: returns a nondeterministic `liquidation_bonus_rate` such that `user_ltv` + `liquidation_bonus_rate` is less than `Fraction::ONE`, and the `liquidation_bonus_rate` being non-zero. Cargo feature: certora-calculate-liquidation-partial-summary

- `max_liquidatable_borrowed_amount_summary`: returns a nondeterministic fraction greater than `Fraction::ONE`. Cargo feature: certora-calculate-liquidation-partial-summary
- `calculate_liquidation_partial_summary`: same as `calculate_liquidation` but `get_liquidation_params` and `max_liquidatable_borrowed_amount` are replaced by their summaries. Cargo feature: certora-calculate-liquidation-partial-summary
- `calculate_liquidation_full_summary`: returns a `CalculateLiquidateResult` such that `borrow_amount * withdraw_amount <= collateral_amount * settle_amount`. This summary is necessary to prove the property "liquidate_obligation reduces LTV". Cargo feature: certora-calculate-liquidation-full-summary
- `liquidate_obligation_full_summary`: same as `liquidate_obligation` but `calculate_liquidation` is replaced by `calculate_liquidation_full_summary`. Cargo feature: certora-calculate-liquidation-full-summary
- `liquidate_obligation_partial_summary`: same as `liquidate_obligation` but `calculate_liquidation` is replaced by `calculate_liquidation_partial_summary`. Cargo feature: certora-calculate-liquidation-partial-summary
- `liquidate_obligation`: original customer code.

- For each rule description, we describe which of the above versions of the function under verification is used.
- To verify the property "liquidate_obligation reduces LTV", we proceed in two steps.
    - The first step is to verify this property over `liquidate_obligation_full_summary`. That is, we summarize `calculate_liquidation` to return a result that satisfies the condition described above for `calculate_liquidation_full_summary`. This is captured by `rule_liquidate_obligation_reduces_ltv`.

- The second step is to ensure that `calculate_liquidation` indeed returns a result that satisfies the necessary condition. This is captured by the rule `rule_calculate_liquidation_verify_summary.` This rule in turn relies on a summary for calculate_liquidation_amounts, which is verified by rule `rule_calculate_liquidation_amounts_verify_summary.`

Module Properties

P-07. Liquidations of healthy positions must revert	
Status: Verified	Specification: A healthy position can never be liquidated.

Rule Name	Status	Description	Link to rule report
rule_liquidate_obligation_healthy_revert	Verified	*The rule asserts that when liquidate_obligation is called on an obligation that is healthy, it returns an error.*	[Report]()

P-08. Integrity rules	
Status: Verified	Specification: The following rules verify the correct flow of funds for liquidation operations.

Rule Name	Status	Description	Link to rule report
rule_integrity_obligation_repay	Verified	*This rule asserts that the repay method of the Obligation datatype either reduces the borrowed_amount_sf or removes the borrow on full repay.*	[Report Part 1]
rule_integrity_obligation_withdraw	Verified	*This rule asserts that the withdraw method of the Obligation datatype either reduces the deposit_amount or removes the collateral if empty.*	
rule_liquidate_obligation_reduces_borrow_amount	Verified	*This rule asserts that liquidation_obligation_partial_summary reduces the borrow amount.*	[Report Part 2]
rule_liquidate_obligation_reduces_collateral_amount	Verified	*This rule asserts that liquidation_obligation_partial_summary reduces the collateral amount.*	

rule_liquidate_obligation_integrity	Verified	*This rule asserts that liquidation_obligation_partial_summary 1) returns withdraw_amount that is less than the collateral amount; 2) returns settle_amount that is less than the borrow amount; and 3) returns repay_amount that is greater than 0. This means that the liquidator must pay a nonzero amount.*	

P-09. Liquidation reduces LTV	
Status: Verified	Specification: The following rules ensure that liquidations reduce a position's LTV.

Rule Name	Status	Description	Link to rule report
rule_liquidate_obligation_reduces_ltv	Verified	*This rule asserts that liquidate_obligation_full_summary reduces the LTV of the obligation. This rule is proved using the summary of calculate_liquidation.*	Report
rule_calculate_liquidation_verify_summary	Verified	*This rule verifies the summary of calculate_liquidation_partial_summary. We assume that the borrow_amount and collateral_amount are less than 2^35.*	Report

rule_calculate_liquidation_amounts_verify_summary	Verified	*This rule verifies the summary of calculate_liquidation_amounts used in rule_calculate_liquidation_verify_summary.*	[Report]()

P-10. Liquidation is profitable	
Status: Verified	Specification: The rule proves that a liquidation attempt is always profitable (up to a precise, but negligible rounding error). We assume an underapproximation for tractability: the borrow_amount and collateral_amount are less than 2^25.

Rule Name	Status	Description	Link to rule report
rule_liquidate_obligation_profitable	Verified	*The rule asserts that that repay_amount and withdraw_amount satisfy repay_amount <= withdraw_amount + 2, that is a liquidator cannot lose more than 2 lamports due to rounding when liquidating an obligation.*	[Report]()

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.