K-Lend Formal Verification

Security Assessment

October 6th, 2025 — Prepared by OtterSec

Jamie Hill-Daniel jamie@osec.io

Andrew Haberlandt a@osec.io

Table of Contents

01 — Introduction

Summary

OtterSec developed a formal verification framework for Anchor programs based on the Kani verifier, and used it to formally verify an invariant for several K-Lend instructions. K-Lend is Kamino Finance's lending protocol on Solana, built around a unified liquidity market where users can supply assets to earn yield or post collateral to borrow. The system manages positions through on-chain obligation records and controls risk via configurable parameters such as collateral factors, asset tiers, oracle price feeds, and liquidation rules.

This report begins by introducing the framework and describing how to extend it to verify more instructions, including performance considerations.

We then provide some detail on how the framework handles account deserialization internally, and how to tune it for performance and correctness.

Finally, we present our results for verified instructions, along with the caveats used during verification.

Verification

Kani uses model checking to convert an assertion in the program to a SAT problem, which can be passed to a solver (Cadical in this case) to exhaustively verify that a constraint can never be verified.

Attack Surface

The K-Lend protocol's attack surface can be decomposed into three primary categories:

Oracle inputs. Reserves rely on Scope, Switchboard, or Pyth price feeds to determine asset valuations. Malicious or stale feeds propagate directly into collateral and debt valuation, potentially enabling attackers to manipulate solvency calculations or extract value from the protocol through mispriced positions.

Liquidity flows. The borrower-facing instructions (`deposit_obligation_collateral_v2`, `withdraw_obligation_collateral_v2`, `borrow_obligation_liquidity_v2`, and `repay_obligation_liquidity_v2`) adjust vault balances and obligation solvency. Implementation bugs in these handlers could cause vault balance underflows, bypass protocol caps, or misaccount collateral positions, leading to protocol insolvency or loss of user funds.

Autodeleverage and liquidation. Automatic deleveraging and liquidation paths depend on accurate elevation-group debt tracking, timer mechanisms, and price flags. Logic errors in these mechanisms could strand bad debt in the system or block recovery procedures, threatening protocol solvency during market stress.

For the scope of this project, we focus on the liquidity flow attack surface. Specifically, we verify that no state transition during normal liquidity operations can transition an obligation from a healthy state to an unhealthy state, thereby preventing bugs in the core borrowing and lending logic from creating protocol insolvency.

02 — Framework Usage

Setup

Install Kani according to these instructions. Verification was performed with Kani 0.64 (`cargo install –locked kani-verifier@0.64`) and is not guaranteed to work on other versions.

Verification requires a directory setup as follows:

Directory	Repository	Branch
kfarms	github.com/otter-sec/verify-kfarms	formal-verif3
klend	github.com/otter-sec/verify-klend	formal-verif3
pyth-crosschain	github.com/otter-sec/verify-pyth-crosschain	formal-verif3
scope	github.com/otter-sec/verify-kamino-scope	formal-verif3
solana-verify	github.com/otter-sec/solana-verify	formal-verification-klend

As well as a `run.sh` script to run verification harnesses with optimized options:

```shell
#!/bin/sh
set -eux
cd klend
export RUST_BACKTRACE=1 RUSTFLAGS="-Awarnings"
cargo kani --no-default-features \
    --harness=\ $1 --exact \
    -Z stubbing \
    -Z unstable-options \
    --no-default-checks \
    --no-memory-safety-checks \
    --no-overflow-checks \
    --no-undefined-function-checks \
    --no-unwinding-checks \
    --fail-fast \
    --solver cadical \
    --cbmc-args '--no-standard-checks'
```

Usage

Harnesses

The `#[program]` macro is modified to create verification harnesses for each instruction, in the format `fn verify_<instruction name>` (detailed in 02). This stubs out certain expensive functions and creates symbolic inputs to the instruction. To execute a harness, use (for example), `./run.sh verify_repay_obligation_liquidity_v2`.

`AccountInfo`

Several methods are provided for working with untyped `AccountInfo`. Some common options are

- `as_account::<T>` - Loads a `Ref<T>`, initializing with arbitrary data if not present. Will panic if the account has been initialized with another type.
- `maybe_as_account::<T>` - Loads an `Option<&T>` if already initialized with the correct type.
- `init_as::<T>` - Initializes the account with an arbitrary `T`. Will panic if the account has already been initialized.
- `init_with::<T>` - Initializes the account with a given `T`. Will panic if the account has already been initialized.

Invariants

Accounts may have a 'transition invariant' attached; this is a block of code which receives the account data before and after an instruction is executed and may assert something about the change.

RUST

```rust
#[transition_invariant({
    kani::assert(self.foo == old.foo, "Foo should be unchanged");
    if self.bar ≠ old.bar {
        kani::assert(self.counter > old.counter, "Counter should be incremented if bar changes");
    }
})]
pub struct Account {
    /* ... */
}
```

This may be used to prove that certain state transitions are impossible.

Optimization

Hardware Requirements

As the framework verifies end-to-end and simulates a lot of state, a lot of resources are needed. We use a server with a Ryzen 9950x (verification is single threaded) and 200GB of RAM.

Assert vs Assume

Kani has two important builtin functions:

- `kani::assume(cond)` - Used for preconditions, the solver will abandon a given path state if an `assume` is false.
- `kani::assert(cond, "message")` - Used for invariants, the solver will produce an error message if `cond` can be false.

Error Handling

Error branches can quickly bloat memory usage, so they should be pruned as soon as possible. When encountering an error that will be bubbled back up the start of instruction execution, treat it as a precondition by adding `kani::assume(false)` before returning an error, ensuring the invalid state is abandoned. Please note that the `err!` macro has been modified to automatically insert this statement.

Panicking operations (such as `unwrap` and `expect` or array indexing) cause the program to unwind similarly to bubbling up errors. Consider replacing:

```rust
let variable = function().expect("function should succeed");
```

with:

```rust
let Some(variable) = function() else {
    kani::assert(false, "function should succeed");
    // Or alternatively kani::assume(false), though these errors will not be reported by the
        ↪ verifier
}
```

Usage of `assume` is important for optimizing - as a baseline, adding `kani::assume(false)` to error paths can greatly reduce explosion of state (the `err!` macro has been modified to produce this).

Unwinding and Iterators

To keep a limit on resource usage, our verification harness sets the loop unwinding limit (the number of times a loop may execute before a path is aborted) to 50. If `Not unwinding` appears in the log at a code location, it often indicates that either a loop is unbounded, and should be asserted or assumed to be a sensible limit. The verifier is unable to properly follow the loop semantics - this is common with Rust's iterators.

With this in mind, we have provided many custom functions to replace Rust iterator pipelines in `solana-verify/program/src/verify_helpers.rs` which should be easy to expand on. For example:

```rust
pub fn array_position<const N: usize, T, F>(slice: &[T; N], predicate: F) -> Option<usize>
where
F: Fn(&T) -> bool,
{ /* ... */ }
```

```rust
pub fn array_enumerate_find_mut<const N: usize, T, F>(
slice: &mut [T; N],
predicate: F,
) -> Option<(usize, &mut T)>
where
F: Fn((usize, &mut T)) -> bool,
{ /* ... */ }
```

```rust
pub fn array_all<const N: usize, T, F>(slice: &[T; N], predicate: F) -> bool
where
F: Fn(&T) -> bool,
{ /* ... */ }
```

The key is that simple, C style iterators work best.

```rust
let mut i = 0;
while i < N {  // N is generic over array size
    ...
   i += 1;
}
```

Where possible, iterate over arrays - slices are unbounded length, so our framework asserts a very small `MAX_LEN` of 2 to keep memory manageable. Configuration is explained in 02.

Internals

Account Deserialization/Loading

A key component of our framework, and a large part of the performance overhead, is how account deserialization is handled. Full deserialization support would lead to immense state explosion.

To resolve this, account deserialization is replaced with `kani::any` (instantiating a symbolic account which may be of any type).

Type Constraints

In many cases the account type is known (i.e. when the account is loaded via `AccountLoader`, `InterfaceAccount`, etc.) but untyped `AccountInfos` must be checked dynamically. This is where `assume_types` can be used:

```rust
#[assume_types({
    self.obligation.init_as::<Obligation>();
})]
```

This pre-loads the account with an arbitrary `Obligation`.

Additionally, `remaining_accounts` is a list of 1 or less arbitrary accounts (the limit is defined as `MAX_REMAINING_ACCOUNTS`). Configuration is explained in 02.

To ensure a specific set of accounts is passed to an instruction, use the attribute:

```rust
#[assume_types({
    kani::assume(remaining_accounts.len() == 1);
    remaining_accounts[0].init_with(Reserve {
        liquidity: kani::any(),
        ..Default::default()
    });
})]
```

Harnesses

The harnesses generated inside `#[program]` look like this:

```rust
#[kani::proof]
// Set unwind limit (described above)
#[kani::unwind(50usize)]
// Replace some expensive functions with identical but cheaper ones
#[kani::stub(crate::utils::fraction::U256::div_mod, kamino_lending_stubs::div_mod)]
#[kani::stub(
    crate::lending\_market::farms_ixs::cpi_set_stake_delegated,
    kamino_lending_stubs::cpi_set_stake_delegated
)]
#[kani::stub(core::str::from_utf8, kamino_lending_stubs::str_from_utf8)]
#[kani::stub(std::str::from_utf8, kamino_lending_stubs::str_from_utf8)]
#[kani::stub(core::slice::memchr::memchr, kamino_lending_stubs::memchr)]
pub fn verify_repay_obligation_liquidity_v2() {
    // Arbitrary data provided for instruction paramaters
    let liquidity_amount: u64 = kani::any();
    let conc: anchor_lang::context::ConcreteContext<RepayObligationLiquidityV2> = kani::any();
    // Assume any pre-invariants are true
    conc.to_ctx()
        .accounts
        .__pre_invariants(&ctx.remaining_accounts);
    // Create copies of the data before the instruction runs
    let (accounts, remaining_accounts) = conc.clone_for_transition();
    let result = kamino_lending::repay_obligation_liquidity_v2(ctx, liquidity_amount);
    if !result.is_err() {
        // Run post invariants (including transition invariants)
        conc.to_ctx()
            .accounts
            .__post_invariants(&accounts, &remaining_accounts);
    }
}
```

Configuration Options

There are several limits hard-coded to keep memory usage low.

MAX_LEN

When using our iterator helpers on a slice (as opposed to an array), we assert that the slice is of length <= 2. This can be configured as `MAX_LEN` under `solana-verify/program/src/verify_helpers.rs`.

MAX_REMAINING_ACCOUNTS

Each harness will generate an arbitrary number of `remaining_accounts`, up to and including `MAX_REMAINING_ACCOUNTS` (by default 1). This can be configured under `solana-verify/anchor/src/context.rs`.

03 — Formal Verification

Scope

We model an execution state as a tuple $\sigma = (M, \{R_i\}_{i \in I}, \{O_j\}_{j \in J}, \text{Aux})$, where M is the `LendingMarket`, $\{R_i\}_{i \in I}$ are the reserves in scope, $\{O_j\}_{j \in J}$ are the obligations, and Aux bundles supporting accounts and runtime data (slot, price flags, token accounts, etc.). This abstraction captures the components of the protocol state that are relevant to the instructions we verify.

The verified transition system uses the following instruction set:

$$T = \left\{ \begin{array}{l} \texttt{withdraw\_obligation\_collateral\_v2}, \\ \texttt{deposit\_obligation\_collateral\_v2}, \\ \texttt{borrow\_obligation\_liquidity\_v2}, \\ \texttt{repay\_obligation\_liquidity\_v2} \end{array} \right\}$$

Each transition $\tau \in T$ corresponds to an Anchor instruction handler. A transition $\sigma \xrightarrow{\tau} \sigma'$ mirrors the corresponding handler's logic, including precondition checks and arithmetic operations from the concrete implementation.

To define the property we verify, we first formalize what it means for a state to be "healthy." We define valuation functions for any obligation O based on the protocol market price:

$$V_{\text{dep}}(O) = \sum_{d \in \texttt{O.deposits}} \texttt{d.deposited\_amount} \times \texttt{R}_d\texttt{.liquidity.market\_price\_sf}$$
$$V_{\text{bor}}(O) = \sum_{b \in \texttt{O.borrows}} \texttt{b.borrowed\_amount\_sf} \times \texttt{R}_b\texttt{.liquidity.market\_price\_sf}$$

where $\texttt{R}_d$ is the reserve corresponding to deposit d, and $\texttt{R}_b$ is the reserve corresponding to borrow b. These valuation functions compute the total USD-denominated value of an obligation's deposits and borrows by summing over each position and multiplying by the corresponding reserve's market price.

We adopt the protocol's own solvency check as our health predicate. An obligation O is classified as:

$$\text{Healthy}(O) \iff V_{\text{bor}}(O) \leq V_{\text{dep}}(O)$$
$$\text{Unhealthy}(O) \iff V_{\text{bor}}(O) > V_{\text{dep}}(O)$$

It is important to note that these predicates use the price vector supplied to the instruction at execution time; they do not make statements about future oracle updates or price changes.

Let $\mathcal{S}$ denote the set of all possible states. We define the subset of healthy states as $\mathcal{S}_{\text{healthy}} = \{\sigma \in \mathcal{S} \mid \forall O \in \sigma.\texttt{obligations}, \text{Healthy}(O)\}$. We verify the following property:

$$\forall \tau \in T : (\mathcal{S}_{\text{healthy}} \xrightarrow{\tau} \mathcal{S}_{\text{healthy}})$$

This states that for all transitions τ in T, applying τ to any healthy state yields a healthy state. In other words, if an obligation begins in a healthy state, executing any instruction in T preserves that health property.

More precisely, for every transition $\tau \in T$ and every state transition $\sigma \xrightarrow{\tau} \sigma'$, if the obligation modified by τ is healthy in state σ, then it remains healthy in state σ'. Formally:

$$\forall \tau \in T, \forall \sigma, \sigma' : \left(\sigma \xrightarrow{\tau} \sigma' \land O_\tau \in \sigma.\texttt{obligations} \land \text{Healthy}_\sigma(O_\tau)\right) \implies \text{Healthy}_{\sigma'}(O_\tau)$$

where O_τ denotes the obligation modified by transition τ, and the subscripts on Healthy indicate evaluation in the respective states. This property is enforced in our implementation by the `#[transition_invariant]` attribute attached to the `Obligation` struct in `obligation.rs`, which compares the old and new deposit and borrow values using the valuation functions defined above.

In plain language, we verify that if an obligation is healthy (i.e. the value of the borrows is below that of the collateral), it cannot become unhealthy through execution of verified instructions.

This invariant is expressed as:

```rust
#[transition_invariant({
    fn implies(cond: bool, then: bool) -> bool {
        !cond || then
    }
    let (old_deposits, old_borrows) = old.deposit_borrows_value(remaining_accounts);
    let (new_deposits, new_borrows) = self.deposit_borrows_value(remaining_accounts);
    implies(old_borrows <= old_deposits, new_borrows <= new_deposits)
})]
struct Obligation { /* ... */ }
```

Where `deposit_borrows_value` is defined as follows:

1. Begin by loading the `Reserve` from the provided accounts, asserting that the framework correctly constrained the type.

```rust
impl Obligation {
    pub fn deposit_borrows_value(&self, remaining_accounts: &[AccountInfo]) -> (u128, u128) {
        kani::assert(remaining_accounts.len() == 1, "1 reserve account should be provided");
        let reserve_acc = remaining_accounts[0];
```

```
        kani::assert(reserve_acc.is_initialized(), "reserve not initialized");
        let Some((reserve_key, reserve)) = reserve_acc
            .maybe_as_account::<crate::Reserve>()
            .map(|r| (reserve_acc.key, r))
        else {
            kani::assert(false, "remaining_account is not a Reserve");
            unreachable!()
        };
```

2. Then we iterate over each deposit, `assume`'ing that it refers to our loaded `Reserve`, summing the deposit value.

```rust
// ...
    let mut tot_deposits: u128 = 0;
    for i in 0..self.deposits_count() {
        let d = &self.deposits[i];
        kani::assume(d.deposit_reserve == *reserve_key);

        let value = (d.deposited_amount as
            ↪ u128).checked_mul(reserve.liquidity.market_price_sf);
        kani::assume(value.is_some());

        tot_deposits += value.unwrap();
    }
```

3. Finally we repeat this for the borrows and return the two values.

```rust
// ...
    let mut tot_borrows: u128 = 0;
    for i in 0..self.borrows_count() {
        let b = &self.borrows[i];
        kani::assume(b.borrow_reserve == *reserve_key);
        let value =
            (b.borrowed_amount_sf as u128).checked_mul(reserve.liquidity.market_price_sf);
        kani::assume(value.is_some());

        tot_borrows += value.unwrap();
    }
    (tot_deposits, tot_borrows)
```

Modeling Effort, Assumptions, and Caveats

Realizing the proof in the `klend-verify` harness required substantial restriction of the symbolic state space to ensure tractability for the model checker. The verified invariant holds only under the constrained environment described below.

Reserve configuration

Each harness constrains `remaining_accounts` to contain exactly one `Reserve`, initialized via `init_with` with default padding and an arbitrary `ReserveLiquidity`. The `ReserveLiquidity` has arbitrary values, except that:

- `borrowed_amount_sf` is fixed at 50,
- the oracle spot price lies in the interval $[20, 500]$,
- `mint_decimals` < 19.

These restrictions keep `u128` multiplications tractable for the SAT solver.

Obligation structure

To reduce the simulated state, it is assumed that there exactly two deposits, and two borrows. This is enforced by `kani::assume` gates in helper accessors:

```rust
pub fn deposits_count(&self) -> usize {
    kani::assume(
        self.deposits[0].deposit_reserve != Pubkey::default()
            && self.deposits[1].deposit_reserve != Pubkey::default(),
    );
    2
}

pub fn borrows_count(&self) -> usize {
    kani::assume(
        self.borrows[0].borrow_reserve != Pubkey::default()
            && self.borrows[1].borrow_reserve != Pubkey::default(),
    );
    2
}
```

Fraction arithmetic

The `fixed::types::U68F60` type is wrapped in a `Fraction` newtype to allow controlled operator overloading. Division operations are guarded with solver assertions against zero denominators:

```rust
pub struct Fraction(pub fixed::types::U68F60);
```

This allows certain operations to be modified to be more easily optimized by the solver. In particular, division is asserted not to be by zero - this allows complex panic unwinding logic to be elided.

```rust
impl Div for Fraction {
    type Output = Fraction;
    fn div(self, rhs: Self) -> Self::Output {
        kani::assert(!rhs.is_zero(), "may not divide by zero");
        Self(self.0 / rhs.0)
    }
}

impl Div<u128> for Fraction {
    type Output = Fraction;
    fn div(self, rhs: u128) -> Self::Output {
        kani::assert(rhs != 0, "may not divide by zero");
        Self(self.0 / rhs)
    }
}
```

Branch pruning and stubbing

Certain code paths are stubbed or disabled to avoid solver blow-ups. For example, the `amount_to_borrow == u64::MAX` branch in `Reserve::calculate_borrow` is disabled, and parts of `update_elevation_group_debt_trackers_on_repay` are stubbed when the elevation group is nonzero. Similarly, expensive calls such as `U256::div_mod` are replaced with simplified stubs.

Infeasible paths

The `err!` macro and other failure cases insert `kani::assume(false)` to cut off infeasible execution traces early.

These assumptions and caveats represent trusted modeling obligations. A comprehensive enumeration and justification of all constraints is given in Section 05.

04 — Results

Using this framework, we have verified the healthy ⇒ healthy invariant for an initial set of instructions. All results must be interpreted under the modeling constraints described in Section 03.

Verified Instructions

withdraw_obligation_collateral_v2

This instruction was verified with no additional caveats in 9045 seconds.

deposit_obligation_collateral_v2

This instruction was verified with no additional caveats in 15730 seconds.

borrow_obligation_liquidity_v2

Under `calculate_borrow`, in `programs/klend/src/state/reserve.rs`, the branch where `amount_to_borrow == u64::MAX` is currently disabled due to symbolic fractional division causing memory state explosion. Verification was completed in 38460 seconds.

repay_obligation_liquidity_v2

Under `update_elevation_group_debt_trackers_on_repay`, in `programs/klend/src/lending_market/lending_operations.rs`, the branch where `obligation.elevation_group ≠ ELEVATION_GROUP_NONE` is currently disabled due to complex iteration and indexing causing memory state explosion. Verification was completed in 21500 seconds.

05 — Appendix A

Assumptions

The following `kani::assume` statements were inserted throughout the codebase to constrain the symbolic state space:

handler_update_lending_market.rs

- Line 266: Assume market name contains only ASCII characters

lending_operations.rs

- Lines 176, 185, 194, 208, 220, 233, 262, 305: Prune infeasible error paths
- Lines 663, 672, 680, 688, 706, 825: Stub expensive computations
- Lines 1637, 1649, 1658, 1664, 1787: Disable edge case branches
- Line 2595: Bound deposit index to array length
- Line 2596: Limit maximum deposits to 8
- Line 2670: Bound borrow index to obligation’s borrow array
- Lines 2672, 2676, 2684: Stub elevation group logic
- Lines 3044, 3057, 3067: Prune unreachable paths
- Lines 3316, 3322: Disable specific error conditions
- Lines 3389, 3390: Limit tier arrays to maximum 2 elements
- Lines 3400, 3406, 3412, 3418, 3424, 3430, 3436: Prune additional error paths

obligation.rs

- Lines 220, 231: Ensure deposit/borrow reserve keys match
- Lines 223, 234: Require valid option values
- Lines 471, 480: Constrain obligation structure
- Line 489: Limit deposits array to 8 elements
- Lines 498, 517: Prune infeasible states

File	Line	Assumption	Reason
handlers/handler_update_lending_market.rs	266	`kani::assume(market.name.iter().all(|c| c.is_ascii()));`	Avoid expensive UTF8 checks
lending_market/lending_operations.rs	176	`kani::assume(false);`	Prune error path early
lending_market/lending_operations.rs	185	`kani::assume(false);`	"
lending_market/lending_operations.rs	194	`kani::assume(false);`	"
lending_market/lending_operations.rs	208	`kani::assume(false);`	"
lending_market/lending_operations.rs	220	`kani::assume(false);`	"
lending_market/lending_operations.rs	233	`kani::assume(false);`	"
lending_market/lending_operations.rs	262	`kani::assume(false);`	"
lending_market/lending_operations.rs	305	`kani::assume(false);`	"
lending_market/lending_operations.rs	663	`kani::assume(false);`	"
lending_market/lending_operations.rs	672	`kani::assume(false);`	"
lending_market/lending_operations.rs	680	`kani::assume(false);`	"
lending_market/lending_operations.rs	688	`kani::assume(false);`	"
lending_market/lending_operations.rs	706	`kani::assume(false);`	"
lending_market/lending_operations.rs	825	`kani::assume(false);`	"
lending_market/lending_operations.rs	1637	`kani::assume(false);`	"
lending_market/lending_operations.rs	1649	`kani::assume(false);`	"
lending_market/lending_operations.rs	1658	`kani::assume(false);`	"
lending_market/lending_operations.rs	1664	`kani::assume(false);`	"
lending_market/lending_operations.rs	1787	`kani::assume(false);`	"

Table 05.1: Modeling assumptions extracted from verification build.

File	Line	Assumption	Reason
lending_market/lending_operations.rs	2595	`kani::assume(i < deposits.len());`	Statically true, used as a hint to the optimizer
lending_market/lending_operations.rs	2596	`kani::assume(i < 8);`	″
lending_market/lending_operations.rs	2670	`kani::assume(obligation_borrow_index < obligation.borrows.len());`	Prevent panic with OOB indexing
lending_market/lending_operations.rs	2672	`kani::assume(false);`	Only `ELEVATION_GROUP_NONE` is currently verified due to memory usage issues
lending_market/lending_operations.rs	3044	`kani::assume(false);`	Prune error path early
lending_market/lending_operations.rs	3057	`kani::assume(false);`	″
lending_market/lending_operations.rs	3067	`kani::assume(false);`	″
lending_market/lending_operations.rs	3316	`kani::assume(false);`	″
lending_market/lending_operations.rs	3322	`kani::assume(false);`	″
lending_market/lending_operations.rs	3389	`kani::assume(deposit_tiers.len() ⩽ 2);`	We verify for a maximum of two borrows and deposits
lending_market/lending_operations.rs	3390	`kani::assume(borrow_tiers.len() ⩽ 2);`	″
lending_market/lending_operations.rs	3400	`kani::assume(false);`	Prune error path early

Table 05.2: Modeling assumptions extracted from verification build.

File	Line	Assumption	Reason
lending_market/lending_operations.rs	3406	kani::assume(false);	"
lending_market/lending_operations.rs	3412	kani::assume(false);	"
lending_market/lending_operations.rs	3418	kani::assume(false);	"
lending_market/lending_operations.rs	3424	kani::assume(false);	"
lending_market/lending_operations.rs	3430	kani::assume(false);	"
lending_market/lending_operations.rs	3436	kani::assume(false);	"

Table 05.3: Modeling assumptions extracted from verification build.

File	Line	Assumption	Reason
state/obligation.rs	220	kani::assume(d.deposit_reserve == *reserve_key);	For verification purposes, it is assumed that all deposits and borrows use the same single reserve
state/obligation.rs	231	kani::assume(b.borrow_reserve == *reserve_key);	″
state/obligation.rs	223	kani::assume(value.is_some());	Assume multiplication does not overflow to avoid a panic
state/obligation.rs	234	kani::assume(value.is_some());	″
state/obligation.rs	471	kani::assume(...)	We verify for a maximum of two borrows and deposits
state/obligation.rs	480	kani::assume(...)	″
state/obligation.rs	489	kani::assume(self.deposits.len() ⩽ 8);	Statically true, used as a hint to the optimizer
state/obligation.rs	498	kani::assume(false);	Avoid generating an invalid enum value
state/obligation.rs	517	kani::assume(false);	″

Table 05.4: Modeling assumptions extracted from verification build.