Formal Verification Report: Deriverse Protocol v1

- Repository: https://github.com/Cyfrin/audit-2025-10-deriverse
- Audit Commit Hash: 30b06d2
- Fixes Commit Hash: 2dde892
- Date: November 2025
- Author: @alexzoid_eth (@CyfrinAudits private formal verification engagement)
- Certora Solana Prover version: 8.5.1

Table of Contents

About Deriverse Protocol

Deriverse is a decentralized exchange (DEX) built on Solana that combines automated market maker (AMM) liquidity pools with central limit order book (CLOB) functionality for both spot and perpetual futures markets.

Spot Trading uses a hybrid architecture where the constant-product AMM (`k = asset_tokens × crncy_tokens`) provides baseline liquidity while limit orders on an RB-tree based order book offer price improvement. Traders can execute market orders, place limit orders, or swap tokens directly against the AMM & CLOB. Liquidity providers earn a share of trading fees distributed proportionally from protocol-collected fees.

Perpetual Futures operates on an isolated margin model with configurable leverage (dynamically capped based on daily volatility). The system includes funding rate mechanisms that periodically rebalance long/short positions, automated margin calls with penalty rates feeding an insurance fund, and a rebalancing mechanism for position management. Traders must purchase a "market seat" at a dynamic price based on current participant count before trading perpetuals; this seat can be sold back when exiting. Perpetual markets are created by upgrading existing spot instruments when they meet readiness criteria.

Tokenomics centers on the DRVS native token, which enables participation in on-chain governance voting (fee rates, discount parameters, pool ratios), earning of dividend distributions from protocol fees. Additionally, users can receive airdrop according to their spot/perp trading and LP activities.

Additional Features include a referral program providing fee discounts to referred traders with rewards to referrers, a points program that accumulates trading/LP activity rewards convertible to DRVS via airdrops, and a private mode for controlled launches with whitelisted participants. All protocol state is stored on-chain through Solana PDAs with no custodial control over user funds.

Formal Verification Approach

A reusable verification framework was developed to streamline property creation and maintenance. The framework provides the following capabilities:

1. Properties as simple Rust functions: Verification rules are written as regular Rust functions, while all prover configuration and boilerplate generation happens under the hood via declarative macros
2. Automatic rule generation by property type: Depending on the category (Valid State, Variable Transitions, State Transitions), rules are automatically generated and tested against all protocol instructions. Additionally, the framework supports targeted rules for specific functions with minimal configuration (High-Level, unit-test style)
3. Script-based config generation: Configuration files for all instructions and properties are generated automatically by a build script, eliminating manual setup

Correct prover operation requires a specific code structure, which is achieved through conditional compilation with the `certora` feature flag. This enables mock implementations and state capture mechanisms without modifying production code.

The verification effort focuses on the voting/governance system as an ideal candidate for isolated verification due to its well-defined boundaries and limited external dependencies.

Formal Verification Methodology

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

Assumptions

While comprehensive state exploration is powerful, it introduces a challenge: the prover can generate semantically invalid states that produce false positive violations. For example, an integer overflow may occur during increment when the prover assumes a variable already holds `MAX_UINT`. The solution is to constrain input data and state values for the prover, but these constraints must be applied carefully to avoid masking real bugs.

We classify assumptions into three categories:

Safe Assumptions

These reflect real-world constraints that don't impact security guarantees:

- Numeric values bounded to practical ranges (e.g., using effective `u128` range instead of `u256`, timestamps within reasonable bounds)
- Protocol constants honored (e.g., minimum voting token threshold is set at initialization and cannot decrease)
- Message sender (signer) assumed distinct from program accounts

Proved Assumptions

These are Valid State invariants that are **first verified by the prover**, then used as assumptions when verifying other properties. This creates a chain of trust where each assumption is backed by a mathematical proof:

- User's token balance cannot exceed total protocol token supply
- User's recorded vote references an existing voting round
- Account discriminators correctly identify account types
- Protocol version consistency across all accounts
- etc

Unsafe Assumptions

These reduce verification scope or work around prover limitations, potentially missing some execution paths. These modifications are implemented via conditional compilation (`#[cfg(feature = "certora")]`) — production code compiles the original implementation, while the `certora` feature flag enables prover-compatible alternatives.

- Mock Clock: Solana's `Clock` sysvar replaced with a symbolic mock that returns nondeterministic but

bounded timestamp/slot values

- Bounded vectors: Dynamic `Vec<T>` replaced with fixed-size arrays (3 elements) to avoid symbolic length issues — affects `BoundedBaseCrncy`, `BoundedAssets`, `BoundedClientCommunityRecords`, `BoundedOperators`
- Early panic attribute: `#[cvlr::early_panic]` added to instruction entry points for proper error path handling
- Simplified `finalize_voting()`: Voting finalization logic modified for prover compatibility
- `DirectStateAccess` trait: Enables direct memory access for state capture, bypassing standard account deserialization
- Public field visibility: Private struct fields exposed as `pub` for verification state inspection
- Signed integer conversion: `i64` fields converted to `NativeInt` with explicit bounds (`POSITIVE_I64_MAX`, `SAFE_BOUND`) due to prover treating signed comparisons as unsigned
- Loop unrolling: Capped at 3 iterations for array processing

The Certora Solana Prover is under active development. As the prover matures, some of these workarounds may become unnecessary and can be removed from the codebase. Additionally, support for certain complex instructions (such as `swap` or `deposit`) was not implemented as they would require an excessive amount of conditional compilation modifications, making maintenance impractical.

Types of Properties

When constructing properties in formal verification, we deal with five types according to Certora's official classification. The first three (Valid State, Variable Transitions, State Transitions) are verified parametrically across all external instructions, while High-Level and Unit Tests target specific functions.

Valid State (`valid_state.rs`)

Conditions that MUST always remain true throughout the protocol's lifecycle. The verification process defines valid state constraints, executes an external instruction, then confirms the invariant still holds.

Single-state invariants (`state_inv_*`) apply constraints on individual account types, while cross-state invariants (`state_inv_cross_*`) span multiple account types.

Example: "Protocol fee rates must be within governance bounds"

Variable Transitions (`variable_transitions.rs`)

Verify that specific storage variables change only under expected conditions. The process captures a variable value before instruction execution, runs the instruction, then asserts the variable changed only as permitted or remained unchanged.

Example: "Protocol addresses (DRVS authority, USDC mint) can never change after initialization"

State Transitions (`state_transitions.rs`)

Flexible checks for specific behaviors or conditions. These apply preconditions via `assume_pre()`, execute the instruction, then assert postconditions via `assert_post()`.

Example: "Only operators can change protocol settings"

High-Level (`high_level.rs`)

Business logic properties verified against specific instructions rather than parametrically across all instructions. These target complex scenarios that require custom setup and assertions.

Example: "Vote with zero tokens has no effect on accumulators"

Unit Tests (`unit_test.rs`)

Instruction-specific tests verifying expected behavior, revert conditions, and edge cases. Unlike invariants which verify properties across all instructions, unit tests focus on individual instruction semantics.

Example: "voting() reverts when user already voted in current round"

Verification Properties

Links to specific Certora Prover runs are provided for each property, with status indicators.

Valid State

Valid State properties define the fundamental invariants that must always hold true throughout the protocol's lifecycle. These properties are organized by account type and proven as invariants.

Root Account Invariants

Property	Name	Description	Status
VS-R-01	`root_account_valid`	Root account has correct type identifier (TAG = ROOT)	✅
VS-R-02	`referral_rewards_within_limits`	Referral discounts and ratios are within allowed limits	✅
VS-R-03	`protocol_has_clients`	Protocol always has at least one registered client	✅

Client Primary Account Invariants

Property	Name	Description	Status
VS-CP-01	`referral_rewards_within_limits`	Client account has correct type identifier (TAG = CLIENT_PRIMARY)	✅
VS-CP-02	`client_referral_rewards_within_limits`	Client's referral rewards are within allowed limits	✅
VS-CP-03	`user_cannot_refer_themselves`	User cannot set themselves as their own referrer	✅
VS-CP-04	`client_update_timestamp_valid`	Client's last update is not in the future	✅

Community Account Invariants

Property	Name	Description	Status
VS-C-01	`community_account_valid`	Community account has correct type identifier (TAG = COMMUNITY)	✅
VS-C-02	`protocol_fees_within_limits`	Protocol fee rates and parameters are within governance bounds	✅
VS-C-03	`voting_round_started_in_past`	Voting round cannot start in the future	✅
VS-C-04	`no_votes_before_first_round`	All voting fields are zero before first round starts	✅
VS-C-05	`active_round_has_deadline`	If voting round exists, it has a valid deadline	✅

Client Community Account Invariants

Property	Name	Description	Status
VS-CC-01	`client_community_account_valid`	Client community account has correct type identifier	✅
VS-CC-02	`user_token_balances_valid`	User's token balances are non-negative and within valid range	✅
VS-CC-03	`user_sync_timestamp_valid`	User's last synchronization is not in the future	✅
VS-CC-04	`user_vote_timestamp_valid`	User's last vote timestamp is not in the future	✅
VS-CC-05	`uninitialized_user_has_no_tokens`	User without sync history has no governance tokens	✅

Holder Account Invariants

Property	Name	Description	Status
VS-H-01	`holder_account_valid`	Holder account has correct type identifier (TAG = HOLDER)	✅

Token Account Invariants

Property	Name	Description	Status
VS-T-01	`token_account_valid`	Token account has correct type identifier (TAG = TOKEN)	✅

Instrument (Market) Account Invariants

Property	Name	Description	Status
VS-I-01	`market_account_valid`	Market account has correct type identifier (TAG = INSTR)	✅
VS-I-02	`market_has_distinct_tokens`	Market's base and quote tokens must be different	✅
VS-I-03	`token_decimals_valid`	Token decimal counts are within valid range (<32)	✅
VS-I-04	`perp_open_interest_valid`	Perpetual open interest is non-negative and within valid range	✅
VS-I-05	`market_risk_params_valid`	Market risk parameters (variance, volatility, leverage) are within bounds	✅
VS-I-06	`market_update_timestamp_valid`	Market's last update is not in the future	✅

Spot Trade Account Invariants

Property	Name	Description	Status
VS-ST-01	`spot_trade_account_valid`	Spot trade account has correct type identifier	✅
VS-ST-02	`spot_trade_timestamp_valid`	Spot trade timestamp is not in the future	✅

Perp Trade Account Invariants

Property	Name	Description	Status
VS-PT-01	`perp_trade_account_valid`	Perp trade account has correct type identifier	✅
VS-PT-02	`perp_trade_timestamp_valid`	Perp trade timestamp is not in the future	✅

Candles Account Invariants

Property	Name	Description	Status
VS-CA-01	`candles_account_valid`	Candles account has correct type identifier	✅
VS-CA-02	`candles_timestamp_valid`	Candles timestamp is not in the future	✅

Private Client Account Invariants

Property	Name	Description	Status
VS-PC-01	`private_client_account_valid`	Private client account has correct type identifier (or uninitialized)	✅

Cross-State Invariants: Balance

Property	Name	Description	Status
VS-X-01	`community_balances_valid`	Community token balances are non-negative and within valid range	✅
VS-X-02	`previous_round_votes_within_supply`	Previous round votes cannot exceed token supply at that time	✅
VS-X-03	`votes_cannot_exceed_supply`	Total votes cast cannot exceed available token supply	✅
VS-X-04	`base_currencies_within_limit`	Base currency count doesn't exceed token count	✅

Cross-State Invariants: ID Validation

Property	Name	Description	Status
VS-X-05	`user_id_valid`	User ID is within valid range	✅
VS-X-06	`token_id_valid`	Token ID is registered in protocol	✅
VS-X-07	`market_id_valid`	Market ID is registered in protocol	✅
VS-X-08	`market_tokens_valid`	Market's base and quote tokens are registered	✅
VS-X-09	`spot_order_ids_valid`	Spot order references valid market and tokens	✅
VS-X-10	`perp_order_ids_valid`	Perp order references valid market and tokens	✅
VS-X-11	`user_referral_links_valid`	User's referral links reference valid referrers	✅
VS-X-12	`user_assets_within_limit`	User's asset count doesn't exceed available tokens	✅
VS-X-13	`candles_market_valid`	Candles data references valid market	✅
VS-X-14	`referrer_id_valid`	Referrer ID is a valid user	✅

Cross-State Invariants: Account Consistency

Property	Name	Description	Status
VS-X-15	`community_account_matches_user`	Community account belongs to correct user	✅
VS-X-16	`user_community_state_valid`	User's community state references valid round	✅

Property	Name	Description	Status
VS-X-17	`all_accounts_same_version`	All protocol accounts use same version	✅
VS-X-18	`private_client_registration_valid`	Private client entries have valid timestamps	✅

Cross-State Invariants: Voting

Property	Name	Description	Status
VS-X-19	`user_voted_in_existing_round`	User's recorded vote is in a round that exists	✅
VS-X-20	`user_synced_to_existing_round`	User is synchronized to a round that exists	✅
VS-X-21	`user_tokens_within_protocol_total`	User cannot have more tokens than exist in protocol	✅
VS-X-22	`user_voting_power_within_supply`	User's voting power cannot exceed total supply	✅
VS-X-23	`user_vote_weight_within_supply`	User's vote weight cannot exceed total supply	✅
VS-X-24	`vote_weight_snapshot_at_round_start`	Vote weight is fixed at round start (tokens deposited after don't count)	✅
VS-X-25	`vote_requires_tokens`	User must have tokens to vote	✅
VS-X-26	`vote_has_timestamp`	Every vote has a recorded timestamp	✅
VS-X-27	`no_vote_before_sync`	User cannot vote before synchronizing with round	✅
VS-X-28	`vote_weight_not_inflated`	Vote weight cannot exceed user's available tokens	✅
VS-X-29	`vote_weight_consistent_in_round`	Vote weight stays the same throughout a round	✅
VS-X-30	`deadline_requires_active_round`	Voting deadline only exists for active rounds	✅
VS-X-31	`sufficient_tokens_for_active_voting`	Minimum token threshold met for active voting	✅

Variable Transitions

Variable Transition properties verify that specific storage variables change only under expected conditions.

Root Account Transitions

Property	Name	Description	Status
VT-R-01	`new_entities_one_at_a_time`	New users, tokens, markets, and referral links are added one at a time	✅
VT-R-02	`points_program_cannot_shorten`	Points program expiration can only be extended, never shortened	✅
VT-R-03	`protocol_addresses_unchangeable`	Core protocol addresses (holder, LUT, DRVS mint, operator) cannot change after setup	✅

Client Primary Account Transitions

Property	Name	Description	Status
VT-CP-01	`user_identity_unchangeable`	User's wallet address, LUT, and ID cannot change after registration	✅
VT-CP-02	`user_trades_one_at_a_time`	User's spot, perp, LP trades and assets increase one at a time	✅
VT-CP-03	`user_points_never_decrease`	User's earned points can only increase	✅
VT-CP-04	`user_activity_moves_forward`	User's activity timestamp only moves forward	✅
VT-CP-05	`referral_benefits_preserved`	User's referral links and expirations can only improve	✅

Community Account Transitions

Property	Name	Description	Status
VT-C-01	`voting_rounds_progress_forward`	Voting round counter and base currency count only increase	✅
VT-C-02	`minimum_stake_unchangeable`	Minimum token threshold for voting is set once and never changes	✅
VT-C-03	`fee_changes_gradual`	Protocol fees can only change by ±1 per voting round (no sudden jumps)	✅
VT-C-04	`voting_rounds_never_restart`	Voting round start time only moves forward	✅

Client Community Account Transitions

Property	Name	Description	Status
VT-CC-01	`user_community_id_unchangeable`	User's community account ID cannot change after creation	✅
VT-CC-02	`user_community_sync_moves_forward`	User's community sync timestamp only moves forward	✅
VT-CC-03	`user_voting_progress_forward`	User's voting round tracking only moves forward (no going back)	✅

Holder Account Transitions

Property	Name	Description	Status
VT-H-01	`operators_added_one_at_a_time`	New protocol operators are added one at a time	✅

Token Account Transitions

Property	Name	Description	Status
VT-T-01	`token_config_unchangeable`	Token ID, address, program, mask, and base currency are set once and never change	✅

Instrument (Market) Account Transitions

Property	Name	Description	Status
VT-I-01	`market_config_unchangeable`	Market's tokens, ID, creator, addresses, and decimals cannot change after creation	✅
VT-I-02	`market_activity_grows`	Market's trade counters and derivative count only increase	✅
VT-I-03	`market_history_preserved`	Market's all-time trade counts and timestamp only increase	✅

Spot Trade Account Transitions

Property	Name	Description	Status
VT-ST-01	`spot_order_market_unchangeable`	Spot order's market and token references cannot change	✅
VT-ST-02	`spot_order_timestamp_moves_forward`	Spot order's timestamp only moves forward	✅

Perp Trade Account Transitions

Property	Name	Description	Status
VT-PT-01	`perp_position_market_unchangeable`	Perp position's ID and token references cannot change	✅
VT-PT-02	`perp_position_timestamp_moves_forward`	Perp position's timestamp only moves forward	✅

Candles Account Transitions

Property	Name	Description	Status
VT-CA-01	`candles_market_unchangeable`	Candles' market reference cannot change	✅
VT-CA-02	`candles_data_grows`	Candles timestamp and count only increase	✅
VT-CA-03	`candles_buffer_wraps_correctly`	Candles circular buffer index increases or wraps to zero	✅

State Transitions

State Transition properties verify the correctness of transitions between valid states.

Access Control Properties

Property	Name	Description	Status	Notes
ST-A-01	`only_operator_changes_protocol_settings`	Only the protocol operator can modify admin settings	✅	
ST-A-02	`only_owner_modifies_account`	Only the wallet owner can modify their account data	✅	
ST-A-03	`admin_state_protected`	If any admin-only field changed, the operator must have signed	❌	Issue: #7, #117. ✅ Passed after fix
ST-A-04	`new_private_client_matches_protocol_version`	New private client accounts use the same version as the protocol	✅	
ST-A-05	`private_client_version_match`	When a private client is added, root and private_client header versions must match	❌	Issue: #17. ✅ Passed after fix

Voting System Properties (Transitions)

Property	Name	Description	Status
ST-V-01	`votes_accepted_before_deadline`	Votes are only accepted before the voting deadline	❌
ST-V-02	`user_voting_progress_only_forward`	User's voting round tracking only moves forward	✅
ST-V-03	`user_vote_timestamp_only_increases`	User's vote timestamp only increases	✅
ST-V-04	`total_votes_grow_or_reset`	Total votes can only increase during a round or reset at round end	✅
ST-V-05	`new_round_updates_start_time`	When a new voting round starts, the start time is updated	✅
ST-V-06	`fee_changes_require_new_round`	Protocol fee changes only happen when a new voting round completes	✅
ST-V-07	`voting_power_snapshot_at_round_start`	Total voting power is only recalculated when a new round starts	✅
ST-V-08	`previous_round_results_preserved`	Previous round results are preserved when new round starts	✅
ST-V-09	`new_round_extends_deadline`	Voting deadline extends when new round starts	✅

Voting System Properties (High-Level Rules)

Property	Name	Description	Status
HL-V-01	`vote_rejected_for_wrong_round`	User's vote is rejected if they specify a round that doesn't match the current active round	✅
HL-V-02	`double_voting_rejected`	User cannot submit two votes in the same voting round	✅
HL-V-03	`vote_affects_correct_accumulator`	User's vote actually affects the correct accumulator based on choice	✅
HL-V-04	`voting_is_reachable`	Voting is reachable - user can actually cast a vote	✅

Property	Name	Description	Status
HL-V-05	`no_votes_after_deadline`	After voting deadline passes, no new votes are recorded	✅
HL-V-06	`zero_tokens_no_vote_effect`	User with zero tokens cannot affect vote tallies	✅

Business Logic Properties

Property	Name	Description	Status	Notes
HL-B-01	`user_dividends_synced_with_protocol`	After claiming dividends, user's base currency list matches the protocol's supported currencies	❌	Issue: #80. ✅ Passed after fix
ST-B-01	`votes_accepted_before_deadline`	Voting accumulators only increase when current time <= voting_end_time	❌	Issue: #16. ✅ Passed after fix

Real Issues Properties

This section documents vulnerabilities discovered during the manual audit (by all participants) and formal verification process. Each issue demonstrates how formal properties detected the vulnerability and confirmed its resolution after applying the fix.

[HIGH] Missing Signer Verification in voting_reset Function Allows Unauthorized Execution (#117)

The `voting_reset` function only verifies that the `admin` account's public key matches the operator address, but does not verify that the `admin` account is actually a signer of the transaction. This allows an attacker to execute the `voting_reset` instruction by passing an account that matches the operator address but no signing is required, resetting critical voting parameters without proper authorization.

❌ Violated (in `voting_reset()`): https://prover.certora.com/output/52567/034d832c88f047849b04856c7063192a/?anonymousKey=b7a9454433a41df515c7e8b721fa34ddd9b43b4d

```
// If any admin-only field changed, the operator must have signed
// transitions__admin_state_protected__{instruction}
fn admin_state_protected(
    before: &AllStatesProp,
    after: &AllStatesProp,
    _ctx: &TransitionContext
) -> bool {
    let root_changed = match (&before.root.state, &after.root.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
        _ => false,
    };
    let community_changed = match (&before.community.state, &after.community.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
```

```
        _ => false,
    };
    let instr_changed = match (&before.instrument.state, &after.instrument.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
        _ => false,
    };
    let admin_field_changed = root_changed || community_changed || instr_changed;
    if !admin_field_changed {
        return true; // No admin fields changed, no auth required
    }
    // 1. Admin account is a signer (is_signer=true)
    let is_signer = after.signer_account.is_signer;
    // 2. Admin pubkey matches root_state.operator_address
    let key_matches = match &before.root.state {
        Some(root) => after.signer_account.key == root.operator_address,
        None => false, // Root state required for admin instructions
    };
    is_signer && key_matches
}
```

✅ Passed after the fix (tested with `2.58.3`): https://prover.certora.com/output/52567/745d13da8b5a4a55803e38670010745a/?anonymousKey=b7d2382e683caeaa9c29264477f4c2391c04705d

```
diff --git a/src/program/processor/voting_reset.rs b/src/program/processor/voting_reset.rs
index 4c1de8d..ecbd4a3 100644
--- a/src/program/processor/voting_reset.rs
+++ b/src/program/processor/voting_reset.rs
@@ -55,6 +55,12 @@ pub fn voting_reset(program_id: &Pubkey, accounts: &[AccountInfo]) ->
DeriverseR
         })
     }

+    if !admin.is_signer {
+        bail!(MustBeSigner {
+            address: *admin.key
+        })
+    }
+
     let mut community_state = CommunityState::new(community_acc, program_id,
root_state.version)?;

     let header = community_state.header.upgrade()?;
```

[HIGH] Missing Version Validation for Private Clients Account in new_private_client (#17)

The `new_private_client()` function fails to validate that the provided `private_clients_acc` account matches the protocol version specified in `root_state`. Since different protocol versions use different PDA seeds (which include the version), each version has its own isolated `private_clients_acc`. However, the function does not verify this correspondence, potentially allowing cross-version account misuse.

❌ Violated (in `new_private_client()`): https://prover.certora.com/output/52567/269716a15d4943729d3f503f687108ab/?anonymousKey=c873dc261cd33662659bc5e856520b045c98d4d7

```
// When a private client is added, root and private_client header versions must match
(new_private_client.rs)
// transitions_prop__private_client_version_match__{instruction}
transitions_prop! { private_client_version_match,
    fn assume_pre(states: &AllStatesProp, ctx: &TransitionContext) -> bool {
        true
    }
    fn assert_post(
        before: &AllStatesProp,
        after: &AllStatesProp,
        ctx: &TransitionContext
    ) -> bool {
        let before_has_client = before.private_client.clients[0].has_any_non_zero_field();
        let after_has_client = after.private_client.clients[0].has_any_non_zero_field();
        // Only check if client was added to slot 0
        if before_has_client || !after_has_client {
            return true;
        }
        // Client was added - verify versions match
        match (&before.root.state, &before.private_client.state) {
            (Some(root), Some(pc)) => {
                root.discriminator.version == pc.discriminator.version
            }
            _ => true
        }
    }
}
```

✅ Passed after the fix: https://prover.certora.com/output/52567/ae105778951d4dc09fd2f342dd2e0c0d/?anonymousKey=5fe9ec986b540d2e8e9bca1af24a76e88a8c9353

```
diff --git a/src/program/processor/new_private_client.rs
b/src/program/processor/new_private_client.rs
index b025504..24b74d2 100644
--- a/src/program/processor/new_private_client.rs
+++ b/src/program/processor/new_private_client.rs
@@ -108,6 +108,9 @@ pub fn new_private_client(
         });
     }

+    let _: &mut PrivateClientHeader =
+        FromAccountInfo::from_account_info(private_clients_acc, program_id,
root_state.version)?;
+
     let current_time = Clock::get()
         .map_err(|err| drv_err!(err.into()))?
         .unix_timestamp as u32;
```

[MEDIUM] change_points_program_expiration is permissionless (#7)

The `change_points_program_expiration` function fails to verify that the `admin` account has actually signed the transaction. While the function checks that the provided admin account key matches the expected `operator_address` in the root state, it does not verify that the account is a signer using `admin.is_signer`. Any normal user can pass admin pubkey and execute this instruction without the real admin signing it.

❌ Violated (in `change_points_program_expiration()`): https://prover.certora.com/output/52567/63fe983cbd4542229ef43d7fa5390ca0/?anonymousKey=1d47a4d8fa7642f1cf64cca16b1ef06248881397

```
// If any admin-only field changed, the operator must have signed
// transitions__admin_state_protected__{instruction}
fn admin_state_protected(
    before: &AllStatesProp,
    after: &AllStatesProp,
    _ctx: &TransitionContext
) -> bool {
    let root_changed = match (&before.root.state, &after.root.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
        _ => false,
    };
    let community_changed = match (&before.community.state, &after.community.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
        _ => false,
    };
    let instr_changed = match (&before.instrument.state, &after.instrument.state) {
        (Some(b), Some(a)) => b.admin_fields_changed(a),
        _ => false,
    };
    let admin_field_changed = root_changed || community_changed || instr_changed;
    if !admin_field_changed {
        return true; // No admin fields changed, no auth required
    }
    // 1. Admin account is a signer (is_signer=true)
    let is_signer = after.signer_account.is_signer;
    // 2. Admin pubkey matches root_state.operator_address
    let key_matches = match &before.root.state {
        Some(root) => after.signer_account.key == root.operator_address,
        None => false, // Root state required for admin instructions
    };
    is_signer && key_matches
}
```

✅ Passed after the fix (tested with `2.58.3`): https://prover.certora.com/output/52567/ac4a784a33a94ef59eb4ec703d03a692/?anonymousKey=81fa0cedaf781019c7eb0c898b00ab596e5a19ee

```
diff --git a/src/program/processor/change_points_program_expiration.rs
b/src/program/processor/change_points_program_expiration.rs
index db63d20..1e703b9 100644
--- a/src/program/processor/change_points_program_expiration.rs
+++ b/src/program/processor/change_points_program_expiration.rs
@@ -46,6 +46,12 @@ pub fn change_points_program_expiration(
```

```
     let data = PointsProgramExpiration::new(instruction_data)?;
     let root_state: &mut RootState = RootState::from_account_info(root_acc, program_id)?;

+    if !admin.is_signer {
+        bail!(MustBeSigner {
+            address: *admin.key,
+        });
+    }
+
     if root_state.operator_address != *admin.key {
         bail!(InvalidAdminAccount {
             expected_address: root_state.operator_address,
```

[MEDIUM] Voting is allowed even after voting period's end time (#16)

The `voting()` function does not validate whether the voting period has ended before processing and recording votes. The function only checks the voting end time in the `finalize_voting()` call at the very end, but by that point the user's vote has already been cast and included in the voting tallies. This allows the last user to submit a vote even after the official voting period has expired.

❌ Violated (in `voting()`): https://prover.certora.com/output/52567/f6588e7157604236b166ff7e0d15e1d9/?anonymousKey=1cc62f91c3cebdfb5e7d3508fa47ed2d1354bebb

```
// Voting accumulators only increase when current time <= voting_end_time
// transitions__voting_only_before_end_time__{instruction}
fn voting_only_before_end_time(
    before: &AllStatesProp,
    after: &AllStatesProp,
    _ctx: &TransitionContext
) -> bool {
    let clock_time = get_certora_timestamp();
    match (&before.community.state, &after.community.state) {
        (Some(b), Some(a)) => {
            // If any voting accumulator increased, time must be <= voting_end_time
            let voting_increased =
                a.voting_decr > b.voting_decr ||
                a.voting_incr > b.voting_incr ||
                a.voting_unchange > b.voting_unchange;
            if !voting_increased {
                return true; // No voting occurred, rule trivially holds
            }
            // Voting occurred, so time must have been <= voting_end_time
            clock_time <= b.voting_end_time
        }
        _ => true // No community state, rule trivially holds
    }
}
```

✅ Passed after the fix (tested with `2.58.3`): https://prover.certora.com/output/52567/d8bbaaf0ca5c490b9903111584277a63/?anonymousKey=c2d3ee7a7a39dfc01df3e6507bf0bc8ff9961ba4

```
diff --git a/src/program/processor/voting.rs b/src/program/processor/voting.rs
index 795efc0..25bb64d 100644
--- a/src/program/processor/voting.rs
+++ b/src/program/processor/voting.rs
@@ -105,6 +105,7 @@ pub fn voting(
         let clock = Clock::get().map_err(|err| drv_err!(err.into()))?;
         let time = clock.unix_timestamp as u32;

+        if community_account_header.voting_end_time >= time {
             match data.choice {
                 VoteOption::DECREMENT => {
                     community_account_header.voting_decr += voting_tokens;
@@ -128,6 +129,7 @@ pub fn voting(
             client_community_state.header.last_choice = data.choice as u32;
             client_community_state.header.last_voting_tokens = voting_tokens;
             client_community_state.header.last_voting_time = time;
+        }

         #[cfg(not(feature = "certora"))]
         community_state.finalize_voting(time, clock.slot as u32)?;
```

[MEDIUM] Missing Array Synchronization in dividends_claim Prevents Users from Claiming Dividends for Newly Added Base Currencies (#80)

The `dividends_claim` instruction does not synchronize `client_community_state.data` with `community_state.base_crncy` before processing dividends. If a new base currency is added to the community and a user hasn't performed any operations (like `deposit` or `trading`) that trigger synchronization, the user's `client_community_state.data` array will be shorter than `community_state.base_crncy`. This causes the zip iterator to stop early, preventing users from claiming dividends for newly added base currencies.

❌ Violated (before the fix): https://prover.certora.com/output/52567/0da9e9ee1b91424f892dc1895b79544d/?anonymousKey=80f4cc8482c707b3e2d6277928d96e326e3c953e

```
// After claiming dividends, user's base currency list matches the protocol's supported
currencies
#[rule]
pub fn user_dividends_synced_with_protocol() {
    let program_id = nondet_program_id();
    let accounts = nondet_accounts_dividends_claim();
    log_accounts(&accounts, &mut new_logger());
    let ctx = PropertyContext::new(FunctionName::DividendsClaim, &program_id, &accounts, &
[]);
    let pre = AllStatesProp::capture(&accounts, FunctionName::DividendsClaim);
    pre.log("PRE", &mut new_logger());
    pre.safe_assume_all();
    // Additional constraint for this specific rule
    if let Some(cc) = &pre.client_community.state {
        let client_community_acc = &accounts[4];
        let expected_size = CLIENT_COMMUNITY_ACCOUNT_HEADER_SIZE
            + CLIENT_COMMUNITY_RECORD_SIZE * u64::from(cc.count) as usize;
```

```
        cvlr_assume!(client_community_acc.data_len() == expected_size);
    }
    pre.proved_assume_all(&ctx);
    dividends_claim(&program_id, &accounts).unwrap();
    let post = AllStatesProp::capture(&accounts, FunctionName::DividendsClaim);
    post.log("POST", &mut new_logger());
    if let (Some(community), Some(cc)) = (&post.community.state,
&post.client_community.state) {
        cvlr_assert_eq!(cc.count, NativeInt::from(community.count));
    }
}
```

✅ Passed after the fix (tested with `2.58.8`): https://prover.certora.com/output/52567/ef89a6989ef545ffa3cc58fcb4b6ae70/?anonymousKey=01aef27465f8f97ddb51cfb810ee54a2be7c693f

Unit Tests

Unit tests verify specific behavior of individual instructions, testing exact input/output relationships and revert conditions. Unlike invariants (which verify properties across all instructions), unit tests target one instruction at a time.

Defined in `specs/unit_test.rs`. Added to `confs/unit_tests/` configs.

Voting Unit Tests (9)

Tests for the `voting()` instruction behavior.

Property	Name	Description	Status
UT-V-01	`voting_requires_matching_counter`	voting() requires voting_counter to match community	✅
UT-V-02	`voting_requires_not_already_voted`	voting() requires user hasn't voted yet in current round	✅
UT-V-03	`voting_sets_last_counter_correctly`	voting() sets last_voting_counter to community.voting_counter	✅
UT-V-04	`voting_sets_last_tokens_correctly`	voting() sets last_voting_tokens equal to current_voting_tokens	✅
UT-V-05	`voting_sets_choice_correctly`	voting() sets last_choice to match instruction data	✅
UT-V-06	`voting_accepts_at_exact_deadline`	voting() at exact deadline still accepts vote	✅
UT-V-07	`voting_with_minimum_tokens`	voting() with minimum tokens still affects accumulators	✅
UT-V-08	`voting_syncs_counter_when_slot_old`	voting() syncs current_voting_counter when slot <= voting_start_slot	✅
UT-V-09	`voting_uses_drvs_tokens_when_slot_old`	voting() uses drvs_tokens as voting_tokens when slot <= voting_start_slot	✅

Change Voting Unit Tests (8)

Tests for the `change_voting()` instruction behavior.

Property	Name	Description	Status
UT-CV-01	`change_voting_requires_matching_counter`	change_voting() requires voting_counter to match community	✅
UT-CV-02	`change_voting_requires_has_voted`	change_voting() requires user has voted in current round	✅
UT-CV-03	`change_voting_updates_choice`	change_voting() updates last_choice to new_choice	✅
UT-CV-04	`change_voting_preserves_last_counter`	change_voting() does NOT change last_voting_counter	✅
UT-CV-05	`change_voting_preserves_last_tokens`	change_voting() does NOT change last_voting_tokens	✅
UT-CV-06	`change_voting_moves_between_accumulators`	change_voting() moves tokens from old to new accumulator	✅
UT-CV-07	`change_voting_noop_after_deadline`	change_voting() after deadline doesn't modify accumulators	✅
UT-CV-08	`change_voting_same_choice_allowed`	change_voting() to same choice is allowed (no-op)	✅

Next Voting Unit Tests (8)

Tests for the `next_voting()` instruction behavior.

Property	Name	Description	Status
UT-NV-01	`next_voting_requires_sufficient_tokens`	next_voting() requires drvs_tokens > min_amount	✅
UT-NV-02	`next_voting_increments_counter`	next_voting() increments voting_counter by 1	✅
UT-NV-03	`next_voting_preserves_previous_results`	next_voting() preserves previous round results in prev_* fields	✅
UT-NV-04	`next_voting_resets_accumulators`	next_voting() resets current voting accumulators to zero	✅
UT-NV-05	`next_voting_snapshots_supply`	next_voting() updates voting_supply to current drvs_tokens	✅
UT-NV-06	`next_voting_updates_start_slot`	next_voting() updates voting_start_slot to current slot	✅

Property	Name	Description	Status
UT-NV-07	`next_voting_extends_deadline`	next_voting() extends voting_end_time	✅
UT-NV-08	`next_voting_is_permissionless`	next_voting() is permissionless (any signer can call)	✅

Developer's Guide

This section provides practical guidance for developers working with the formal verification framework.

Installation

For Certora Solana Prover installation, refer to the official documentation.

Requirements:

- Rust toolchain (stable)
- Python 3.8+
- Certora Prover license key (set `CERTORAKEY` environment variable)

Project Setup

The verification setup is based on Certora's solana-spec-template. Below are the integration steps used for this project.

1. Initialize template

```
cd src
git clone https://github.com/Certora/solana-spec-template certora
cd certora
python3 certora-setup.py --workspace /path/to/project --package-name smart-contract --
execute
```

Add the module to `src/lib.rs`:

```
#[cfg(feature = "certora")]
pub mod certora;
```

2. Configure build script

Modify `certora_build.py` to support Solana v2 and skip dev-dependencies:

```
cargo_cmd = ['cargo', 'certora-sbf', '--tools-version', 'v1.43', '--cargo-args=--lib']
```

3. Update Cargo.toml

Add Certora dependencies and feature flag:

```
[features]
certora = ["alpha", "dep:cvlr", "dep:cvlr-solana"]

[workspace.dependencies.cvlr]
version = "0.4.1"

[workspace.dependencies.cvlr-solana]
git = "https://github.com/Certora/cvlr-solana"
branch = "v0.5"

[dependencies.cvlr]
workspace = true
optional = true

[dependencies.cvlr-solana]
workspace = true
optional = true

[package.metadata.certora]
sources = [ "Cargo.toml", "src/**/*.rs" ]
solana_inlining = [ "src/certora/envs/cvlr_inlining_core.txt" ]
solana_summaries = [ "src/certora/envs/cvlr_summaries_core.txt" ]
```

4. Downgrade incompatible crates

The `certora-sbf v1.43` ships with rustc 1.79.0-dev, requiring some dependency downgrades:

```
# indexmap 2.12.x requires rustc 1.82
cargo update indexmap@2.12.1 --precise 2.11.4

# ahash 0.8.12 pulls in getrandom 0.3.4 which doesn't support BPF targets
cargo update ahash --precise 0.8.11
```

Conditional Compilation

All verification-specific code is gated behind the `certora` feature flag. This ensures production builds remain unaffected.

Usage pattern:

```
#[cfg(feature = "certora")]
use crate::certora::mocks::clock::Clock;

#[cfg(not(feature = "certora"))]
use solana_program::clock::Clock;
```

Build commands:

```
# Production build (no verification code)
cargo build-sbf

# Verification build
python3 src/certora/scripts/certora_build.py -l -v

# Or directly with cargo
cargo build --features certora
```

Project Structure

The verification framework is organized under `src/certora/`:

```
src/certora/
├── specs/                    # Property specifications
│   ├── valid_state.rs        # Valid State invariants (state_inv_*, state_inv_cross_*)
│   ├── variable_transitions.rs # Variable Transition properties (variable_prop_*)
│   ├── state_transitions.rs  # State Transition properties (transitions_prop!)
│   ├── high_level.rs         # High-Level rules for specific instructions
│   ├── unit_test.rs          # Unit tests for individual instruction behavior
│   └── base/                 # Shared verification infrastructure
│       ├── mod.rs
│       ├── base.rs           # FunctionName enum, PropertyContext
│       ├── setup.rs          # nondet_accounts_*, setup_* functions
│       ├── state/            # Clone structs, safe_assume(), DirectStateAccess
│       │   ├── mod.rs
│       │   └── {account}.rs  # Per-account Clone structs (root.rs,
community_account_header.rs, etc.)
│       ├── valid_state/      # Property wrappers, AllStatesProp, proved_assume_all()
│       │   ├── mod.rs
│       │   └── {account}.rs  # Per-account property wrappers and macros
│       ├── parametric_rules.rs # Macro-generated rules for all instructions
│       └── log/              # Logging utilities for debugging
│
├── mocks/                    # Prover-compatible replacements
│   ├── mod.rs
│   ├── clock.rs              # Symbolic Clock with bounded timestamp/slot
│   └── bounded_vec.rs        # Fixed-size arrays replacing Vec<T>
│
├── confs/                    # Prover configuration files
│   ├── base.conf             # Base configuration inherited by all others
│   └── properties/           # Generated configs organized by property type
│       ├── valid_state/
│       │   ├── root/         # Root account invariants
│       │   ├── community/    # Community account invariants
│       │   ├── client_primary/
│       │   └── cross/        # Cross-state invariants
│       ├── variable_transitions/
│       │   ├── root/
│       │   ├── community/
│       │   └── ...
│       ├── state_transitions/
```

```
│       ├── high_level/
│       └── unit_tests/
│
├── scripts/                  # Build and generation scripts
│   ├── certora_build.py      # Build script for prover
│   └── generate_confs_all.py # Config generator from spec files
│
└── envs/                     # Prover environment configuration
    ├── cvlr_inlining_core.txt    # Functions to inline
    └── cvlr_summaries_core.txt   # Function summaries
```

The `specs/` directory is organized by property classification type at the top level (matching the five categories described in Types of Properties), while all supporting infrastructure — state capture, account generation, macros, and logging — resides in the `base/` subdirectory.

Key Files

File	Purpose
`specs/base/base.rs`	Defines `FunctionName` enum listing all verified instructions and `PropertyContext` for rule execution
`specs/base/setup.rs`	Contains `nondet_accounts_*()` functions that create symbolic accounts and `setup_*()` helpers for each instruction
`specs/base/state/`	Per-account Clone structs for state capture, `safe_assume()` bounds, `DirectStateAccess` trait
`specs/base/valid_state/`	Per-account property wrappers and macros; `AllStatesProp` struct, `proved_assume_all()`
`specs/base/parametric_rules.rs`	Macros generating rules for each instruction from property definitions
`mocks/clock.rs`	Symbolic `Clock` returning nondeterministic but bounded values via lazy initialization
`mocks/bounded_vec.rs`	`BoundedBaseCrncy`, `BoundedAssets`, etc. — fixed-size arrays (3 elements) replacing dynamic vectors
`scripts/generate_confs_all.py`	Parses spec files and generates `.conf` files for each property × instruction combination
`confs/base.conf`	Base prover settings inherited by all generated configs

Running Verification

Configuration System

Prover configurations are JSON files that specify which rules to verify. The system uses inheritance: all configs extend `base.conf` which contains common settings.

Config generation: The script `scripts/generate_confs_all.py` automatically parses spec files and generates configs for each property × instruction combination. It reads:

- Property definitions from `specs/*.rs` files
- Supported instructions from `specs/base/parametric_rules.rs`

To exclude an instruction from verification, simply comment it out in `parametric_rules.rs`.

```
# Generate all configs to generated_all/ directory
python3 src/certora/scripts/generate_confs_all.py
```

Voting-specific generation: The script `scripts/generate_confs_voting.py` generates configs only for voting-related instructions:

```
VOTING_FUNCS = ["voting", "change_voting", "next_voting"]
```

This is useful for focused verification of the governance system without running unrelated instruction combinations.

```
# Generate configs for voting instructions only
python3 src/certora/scripts/generate_confs_voting.py
```

Current configs: The `src/certora/confs/properties/` directory was also generated by this script, but then manually edited to comment out instructions unrelated to specific invariants. This avoids verifying unnecessary combinations and reduces overall verification time.

Example of a generated config (`valid_state/community/protocol_fees_within_limits.conf`):

```
{
   "override_base_config": "../../../base.conf",
   "build_script": "../../../../scripts/certora_build.py",
   "msg": "state_inv_community__protocol_fees_within_limits",
   "rule": [
      "state_inv_community__protocol_fees_within_limits__fees_deposit",
      "state_inv_community__protocol_fees_within_limits__next_voting",
      "state_inv_community__protocol_fees_within_limits__dividends_allocation",
      "state_inv_community__protocol_fees_within_limits__dividends_claim",
      "state_inv_community__protocol_fees_within_limits__voting",
      // "state_inv_community__protocol_fees_within_limits__deposit",  // commented - not
relevant
      "state_inv_community__protocol_fees_within_limits__voting_reset",
      "state_inv_community__protocol_fees_within_limits__change_voting"
   ]
```

```
}
```

Rule naming convention: `{property_name}__{instruction_name}`

Filtering Generated Configs

The generator supports filtering by properties or instructions:

```
# Generate configs only for specific properties
python3 generate_confs_all.py --filter_props voting_props.txt

# Generate configs only for specific instructions
python3 generate_confs_all.py --filter_funcs voting_funcs.txt

# Both filters
python3 generate_confs_all.py --filter_props props.txt --filter_funcs funcs.txt
```

Filter files use simple format: one name per line, `#` for comments.

Running Verifications

Configs must be run from the directory containing the config file:

```
# Navigate to config directory and run
cd src/certora/confs/properties/valid_state/community
certoraSolanaProver protocol_fees_within_limits.conf
```

Specific rule within a config:

```
cd src/certora/confs/properties/valid_state/community
certoraSolanaProver protocol_fees_within_limits.conf \
  --rule state_inv_community__protocol_fees_within_limits__voting
```

Using generated run scripts (can be run from any directory):

```
# Run all properties in a category
python3 src/certora/confs/properties/valid_state/run_all.py

# Run all properties
python3 src/certora/confs/properties/run_all.py
```

Understanding Results

Prover Execution Flow

When you run the prover, it compiles the code locally and submits it along with source files to a remote server. The server returns a link to the job dashboard where you can monitor progress and view results.

Dashboard: https://prover.certora.com/

Result Statuses

After verification completes, each rule displays one of the following statuses:

Status	Meaning
✅ Verified (green)	Property holds for all possible execution paths
❌ Violated (red)	Counterexample found — property can be broken
⚠️ Timeout	Verification exceeded time limit — consider simplifying the property or adding assumptions
❌ Error	Prover encountered an error (compilation issue, unsupported operation, etc.)

Local vs Remote Execution

The prover can run on Certora's remote servers (default) or locally using CertoraProver. Logs displayed in the remote dashboard are also available in the local console during execution.

Reading Counterexamples

When a property is violated, the prover provides a counterexample (calltrace) showing the exact execution path that breaks the invariant. To debug these, the framework includes logging utilities that capture:

- Input instructions
- All account metadata (keys, signers, writability)
- State field values for each captured account before and after execution

Logging is implemented in `specs/base/log/`.

Adding New Properties

Choosing Property Type

Before creating a property, determine its type based on what you want to verify:

If you want to...	Use	Notes
Verify a condition that must always hold	**Valid State**	Simple, single-account constraints. Also used as assumptions in other properties — keep these short and focused
Track how a single variable changes	**Variable Transitions**	Focus on one variable's allowed mutations
Verify complex state changes with dependencies	**State Transitions**	Multiple variables, cross-account dependencies. Uses all Valid State invariants as assumptions

If you want to...	Use	Notes
Test specific instruction behavior	**High-Level** or **Unit Test**	Targets specific functions, not parametric. High-Level for complex scenarios, Unit Test for edge cases and reverts

How Properties Work Under the Hood

Each property type has a different execution model. Understanding this helps write correct specifications.

Valid State (`state_inv_*`, `state_inv_cross_*`)

Two variants exist:

- `state_inv_*`: Works with a single account type. Only that account's state is initialized with nondeterministic values.
- `state_inv_cross_*`: Works across multiple accounts. All account states are initialized.

Execution flow:

1. Instruction inputs (accounts, parameters) filled with nondeterministic values
2. Pre-state snapshot captured and logged
3. `safe_assume()` applied — bounds on nondeterministic data
4. Instruction executed (prover discards all paths that revert)
5. Post-state snapshot captured and logged
6. Same invariant checked via `cvlr_assert!`

The key insight: Valid State properties use the same constraint for both assumption (pre) and assertion (post). This proves the property is *inductive* — if it holds before, it holds after.

Variable Transitions (`variable_prop_*`)

Execution flow:

1. Nondeterministic setup + pre-state snapshot
2. `safe_assume()` + `proved_assume()` for the specific account being tested
3. Instruction executed
4. Post-state snapshot
5. Post-block receives `(before, after)` states — verify how the variable changed

Example check: "variable only increases or stays the same"

```
fn assert_post(before: &State, after: &State) -> bool {
    after.counter >= before.counter
}
```

State Transitions (`transitions_prop!`)

Similar to Variable Transitions but with two key differences:

1. Pre-block applies ALL Valid State invariants as assumptions (not just for one account)

2. Two macro variants:
    - Full macro: separate `assume_pre()` and `assert_post()` blocks
    - Simplified macro: only `assert_post(before, after)` block

Execution flow:

1. Nondeterministic setup + pre-state snapshot
2. `safe_assume_all()` — bounds on all accounts
3. `proved_assume_all()` — all Valid State invariants as assumptions
4. Instruction executed
5. Post-state snapshot
6. `assert_post(before, after)` — verify transition correctness

High-Level & Unit Tests (`#[rule]`)

Direct rule functions targeting specific instructions. No parametric expansion — you explicitly call the instruction you want to test.

```
#[rule]
pub fn my_test() {
    // 1. Setup
    let (ctx, accounts, pre) = setup_voting();

    // 2. Assumptions
    pre.safe_assume_all();
    pre.proved_assume_all(&ctx);
    cvlr_assume!(/* additional constraints */);

    // 3. Execute
    voting(&ctx.program_id, &accounts, &ctx.instruction_data).unwrap();

    // 4. Capture post-state
    let post = AllStatesProp::capture(&accounts, FunctionName::Voting);

    // 5. Assert
    cvlr_assert!(/* expected condition */);
    // or cvlr_satisfy!() for reachability
}
```

Step-by-Step: Adding a Valid State Property

1. Open `specs/valid_state.rs`
2. Add property function inside appropriate macro block:

```
state_inv_community! {
    // Existing properties...

    // New property: voting deadline must be in the future during active round
    fn voting_deadline_in_future(_ctx: &PropertyContext, s: &CommunityAccountHeaderClone) ->
bool {
        let clock_time = get_certora_timestamp();
        s.voting_counter == NativeInt::from(0u64) ||
        s.voting_end_time > NativeInt::from(clock_time)
    }
}
```

3. Regenerate configs:

```
python3 src/certora/scripts/generate_confs_all.py
```

4. Run verification:

```
cd src/certora/confs/generated_all/valid_state/community
certoraSolanaProver voting_deadline_in_future.conf
```

Step-by-Step: Adding a State Transition Property

1. Open `specs/state_transitions.rs`
2. Add property using the macro:

```
transitions_prop! { voting_increases_total_votes,
    fn assume_pre(_states: &AllStatesProp, _ctx: &TransitionContext) -> bool {
        true // No additional assumptions
    }

    fn assert_post(
        before: &AllStatesProp,
        after: &AllStatesProp,
        _ctx: &TransitionContext
    ) -> bool {
        match (&before.community.state, &after.community.state) {
            (Some(b), Some(a)) => {
                // Total votes can only increase or stay the same
                let before_total = b.voting_incr + b.voting_decr + b.voting_unchange;
                let after_total = a.voting_incr + a.voting_decr + a.voting_unchange;
                after_total >= before_total
            }
            _ => true
        }
    }
}
```

3. Regenerate configs and run.

Step-by-Step: Adding a Unit Test

1. Open `specs/unit_test.rs`
2. Add the rule:

```
#[rule]
pub fn voting_with_max_tokens() {
    let (ctx, accounts, pre) = setup_voting();

    // Assume user has maximum possible tokens
    if let Some(cc) = &pre.client_community.state {
        cvlr_assume!(cc.current_voting_tokens == NativeInt::from(u64::MAX / 2));
    }

    pre.safe_assume_all();
    pre.proved_assume_all(&ctx);

    voting(&ctx.program_id, &accounts, &ctx.instruction_data).unwrap();

    // Verify no overflow occurred
    let post = AllStatesProp::capture(&accounts, FunctionName::Voting);
    if let Some(community) = &post.community.state {
        cvlr_assert!(community.voting_incr < NativeInt::from(u64::MAX));
    }
}
```

3. Add rule name to `confs/properties/unit_tests/voting.conf`:

```
{
   "rule": [
      "voting_with_max_tokens",
      // ... other rules
   ]
}
```

4. Run:

```
cd src/certora/confs/properties/unit_tests
certoraSolanaProver voting.conf --rule voting_with_max_tokens
```

Maintaining Specifications

This section describes what changes are needed in the verification framework when the protocol code evolves.

When to Update Specifications

Event	Action Required
New instruction added	Update `parametric_rules.rs`, `setup.rs`, regenerate configs

Event	Action Required
Instruction removed	Update `parametric_rules.rs`, remove setup function, regenerate configs
Instruction accounts changed	Update `nondet_accounts_*()` and `setup_*()` in `setup.rs`
Instruction parameters changed	Update `setup_*()` function, possibly `PropertyContext`
Account struct fields added/removed	Update Clone struct in `state/{account}.rs`, update `safe_assume()`, update logging
Account struct field types changed	Update Clone struct, `safe_assume()`, logging, possibly invariants
New account type introduced	Create Clone struct, add to `AllStatesProp`, add logging, add invariants
Business logic changed	Review and update affected properties

Files to Modify

File	Responsibility	When to Modify
`specs/base/base.rs`	`FunctionName` enum, `PropertyContext`	New/removed instruction
`specs/base/parametric_rules.rs`	List of instructions for parametric verification	New/removed instruction
`specs/base/setup.rs`	`nondet_accounts_*()`, `setup_*()` functions	New instruction, account changes
`specs/base/state/`	Clone structs, `safe_assume()`, `DirectStateAccess`	Field changes, new account types
`specs/base/valid_state/`	`AllStatesProp`, `proved_assume_all()`	New account types, new invariants to assume
`specs/base/log/`	Logging functions for state and accounts	Field changes, new account types
`specs/valid_state.rs`	Valid State invariants	Field changes, new constraints
`specs/variable_transitions.rs`	Variable Transition properties	Field changes, new immutability requirements
`specs/state_transitions.rs`	State Transition properties	Logic changes, new transition rules
`specs/high_level.rs`	High-Level rules	Business logic changes

File	Responsibility	When to Modify
`specs/unit_test.rs`	Unit tests	Instruction behavior changes
`mocks/bounded_vec.rs`	Bounded vector types	New dynamic arrays in accounts

Step-by-Step: Adding a New Instruction

Example: Adding `withdraw_fees` instruction.

1. Update `base.rs` — add to `FunctionName` enum:

```
pub enum FunctionName {
    // ... existing variants
    WithdrawFees,
}
```

2. Update `parametric_rules.rs` — add instruction to the list:

```
parametric_rules! {
    // ... existing instructions
    [withdraw_fees, WithdrawFees, nondet_accounts_withdraw_fees],
}
```

3. Update `setup.rs` — add account generator and setup function:

```
pub fn nondet_accounts_withdraw_fees<'a, 'info>() -> [AccountInfo<'info>; 5] {
    [
        nondet_account_info(),  // signer
        nondet_account_info(),  // root_acc
        nondet_account_info(),  // community_acc
        nondet_account_info(),  // treasury_acc
        nondet_account_info(),  // system_program
    ]
}

pub fn setup_withdraw_fees<'a, 'info>() -> (PropertyContext, [AccountInfo<'info>; 5],
AllStatesProp) {
    let program_id = nondet_program_id();
    let accounts = nondet_accounts_withdraw_fees();
    let instruction_data: Vec<u8> = vec![]; // or nondet if instruction has parameters

    let ctx = PropertyContext::new(
        FunctionName::WithdrawFees,
        &program_id,
        &accounts,
        &instruction_data,
    );
```

```
    let pre = AllStatesProp::capture(&accounts, FunctionName::WithdrawFees);
    pre.safe_assume_all();

    (ctx, accounts, pre)
}
```

4. Regenerate configs:

```
python3 src/certora/scripts/generate_confs_all.py
```

5. Verify existing properties still pass with new instruction.

Step-by-Step: Adding a New Field to Account Struct

Example: Adding `last_withdrawal_time: u32` to `CommunityAccountHeader`.

1. Update Clone struct in `state/community_account_header.rs`:

```
pub struct CommunityAccountHeaderClone {
    // ... existing fields
    pub last_withdrawal_time: NativeInt,
}
```

2. Update `safe_assume()` in the Clone impl:

```
fn safe_assume(&self) {
    // ... existing assumptions
    let clock_time = get_certora_timestamp();
    cvlr_assume!(self.last_withdrawal_time <= NativeInt::from(clock_time));
}
```

3. Update capture logic (if using `DirectStateAccess`):

```
impl DirectStateAccess for CommunityAccountHeaderClone {
    fn capture(acc: &AccountInfo) -> Option<Self> {
        // ... existing field captures
        last_withdrawal_time: NativeInt::from(header.last_withdrawal_time),
    }
}
```

4. Update logging in `specs/base/log/`:

```
impl CommunityAccountHeaderClone {
    pub fn log(&self, prefix: &str, logger: &mut CvlrLogger) {
        // ... existing field logging
        logger.log_u64(&format!("{} community.last_withdrawal_time", prefix),
self.last_withdrawal_time.into());
    }
}
```

5. Add relevant invariants in `valid_state.rs`:

```
state_inv_community! {
    // Withdrawal timestamp cannot be in the future
    fn withdrawal_timestamp_valid(_ctx: &PropertyContext, s: &CommunityAccountHeaderClone) -
> bool {
        let clock_time = get_certora_timestamp();
        s.last_withdrawal_time <= NativeInt::from(clock_time)
    }
}
```

6. Add variable transition if field has immutability constraints:

```
variable_prop_community! {
    // Withdrawal timestamp only moves forward
    fn withdrawal_time_moves_forward(before: &CommunityAccountHeaderClone, after:
&CommunityAccountHeaderClone) -> bool {
        after.last_withdrawal_time >= before.last_withdrawal_time
    }
}
```

7. Regenerate configs and verify.

Common Issues & Debugging

Typical Problems

1. Prover Errors (Solana Prover Limitations)

The Certora Solana Prover doesn't support all Rust/Solana patterns. This is the most challenging issue to resolve.

Solutions:

- Rewrite the problematic code section using conditional compilation (`#[cfg(feature = "certora")]`)
- Adjust prover configuration parameters
- Add function summaries in `cvlr_summaries_core.txt`
- Add functions to inline list in `cvlr_inlining_core.txt`

2. Timeouts

The prover exceeds time limits when exploring too many execution paths.

Solutions:

- Simplify the property — break complex invariants into smaller ones
- Add more assumptions to reduce state space
- Reduce loop unrolling iterations
- Comment out unrelated instructions in the config file

3. False Positives

The prover finds a "violation" using a state that can never occur in practice. This happens when nondeterministic data isn't properly constrained.

Example: Integer overflow during increment because the prover assumes the variable already holds `MAX_UINT`.

Solutions:

- Add bounds in `safe_assume()` for the affected field
- Add cross-state constraints in `state_inv_cross_*` invariants
- Ensure all nondeterministic instruction data is bounded

4. Vacuity

Since the prover discards all execution paths that revert, it's possible that no valid path remains. The rule passes without actually executing the function — a meaningless result.

Common causes:

- Contradictory assumptions (e.g., `x > 100` and `x < 50`)
- Over-constrained state that fails validation checks
- Missing account initialization that causes early revert

Detection: Use sanity configs to verify that the function can execute at least once with all assumptions applied:

```
cd src/certora/confs/generated_all
certoraSolanaProver sanity.conf
```

The sanity check uses `cvlr_satisfy!(true)` after function execution — if this fails, no valid execution path exists.

Debugging Workflow

1. **Check the calltrace** in the prover dashboard — identify which execution path caused the violation
2. **Look for garbage values** — nondeterministic data without proper bounds appears as unexpected large numbers or invalid states
3. **Add logging** to capture state at key points:

```
let mut logger = new_logger();
log_accounts(&accounts, &mut logger);
pre.log("PRE", &mut logger);

// ... instruction execution ...

post.log("POST", &mut logger);
```

4. **Isolate the issue** — run with a single instruction to narrow down the problem:

```
certoraSolanaProver config.conf --rule property_name__specific_instruction
```

5. **Check assumptions** — verify that `safe_assume()` covers all fields and `proved_assume_all()` is called

for State Transitions

Logging Infrastructure

The framework provides logging utilities in `specs/base/log/`:

Function	Purpose
`new_logger()`	Create a new logger instance
`log_accounts(&accounts, &mut logger)`	Log all account metadata (keys, signers, writability, data length)
`state.log(prefix, &mut logger)`	Log all captured state fields with a prefix ("PRE" or "POST")

Logs appear in:

- Local console during prover execution
- Remote dashboard under "CVL2_*" entries in the calltrace