Formal Verification Report: predict.fun YieldBearingConditionalTokens

- Repository: https://github.com/Cyfrin/audit-2026-01-predict-dot-fun
- Latest Commit Hash: ec9b1f8
- Date: January 2026
- Author: @alexzoid (@cyfrin private formal verification engagement)
- Certora Prover version: 8.6.3

Table of Contents

About predict.fun

predict.fun is a prediction market protocol built on the Gnosis Conditional Token Framework (CTF). Users stake collateral to mint conditional outcome tokens representing positions in binary prediction markets. The protocol extends the standard CTF with Venus Protocol integration for yield generation on idle collateral.

`YieldBearingConditionalTokens` is the core contract combining conditional token logic with Venus yield-bearing functionality. It inherits `WhitelistedERC1155` (transfer-controlled ERC-1155) and `Venus` (yield management via vTokens). This contract and its inherited contracts constitute the verification scope.

Key operations:

- `prepareCondition` / `reportPayouts` — lifecycle management for prediction conditions
- `splitPosition` / `mergePositions` / `redeemPositions` — conditional token operations backed by collateral
- `connectVTokenToUnderlying` / `enableUnderlying` / `disableUnderlying` / `claimYield` — Venus yield integration for idle collateral

When yield is enabled for an underlying token, the contract deposits collateral into Venus (minting vTokens) and redeems it on demand. The `depositedAmount` mapping tracks principal amounts, while yield accrues via vToken exchange rate appreciation.

Formal Verification Methodology

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

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

Types of Properties

Properties are categorized following the official Certora methodology. Valid State, Variable Transitions and State Transitions properties are **parametric** — they are automatically verified against every external function in the contract, including functions added after the specification is completed. High-Level properties target specific function sequences.

Valid State — System-wide invariants that MUST always hold true. These properties define the fundamental constraints of the protocol, such as accounting consistency and structural integrity. Once proven, these invariants serve as trusted assumptions in other properties via `requireInvariant`, reducing verification complexity.

Variable Transitions — Properties that verify 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.

State Transitions — Properties that verify the correctness of transitions between valid states. Building upon the valid state invariants, these properties ensure the protocol's state machine operates correctly and that state changes are both authorized and sequentially valid.

High-Level — Complex multi-step properties verifying business logic integrity. Unlike parametric properties, these target specific function sequences to validate end-to-end protocol behavior.

Unit Tests — Properties that verify basic behavior of individual functions: revert conditions, direct effects on state, and non-effects on unrelated state. Unlike parametric properties, each unit test targets a specific function call.

Verification Process

1. **Setup phase**: Define ghost variables, storage hooks, and helper definitions to model contract state in CVL. Establish the verification harness and configuration. This phase also addresses several prover limitations:
    - Bitwise operations (`bitops.spec`): The prover overapproximates bitwise ops, producing spurious counterexamples. All inline bitwise expressions were extracted into internal helper functions and replaced with precise CVL lookup-table summaries (see Source Code Modifications).
    - ERC-1155 internal arrays (`erc1155.spec`): The prover's pointer analysis crashes on `ERC1155._asSingletonArrays()`. The internal `_burn`, `_mint`, and `_safeTransferFrom` functions are summarized in CVL, bypassing the problematic array construction entirely and tracking balances via ghost variables.

- CTHelpers hash functions (`ct_helpers_lib.spec`): Keccak256-based ID computations (`getConditionId`, `getCollectionId`, `getPositionId`) are opaque to the prover. They are replaced with ghost mappings constrained by axioms that capture essential hash properties (non-zero results, injectivity, commutativity).
- ERC-20 model (`erc20.spec`): Real ERC-20 contracts cause timeouts due to implementation complexity and unbounded address space. A lightweight ghost-variable-based model summarizes all external ERC-20 calls (`balanceOf`, `transfer`, `transferFrom`, `approve`) over a bounded account set, enabling tractable solvency invariants.
- Venus vToken model (`venus.spec`): Real Venus vToken contracts introduce non-linear arithmetic via variable exchange rates, causing solver timeouts. All external vToken calls (`mint`, `redeem`, `redeemUnderlying`, `balanceOfUnderlying`, `exchangeRateCurrent`) are summarized in CVL with a fixed 1:1 exchange rate. Ghost mappings with injectivity and anti-circularity axioms model the vToken-to-underlying relationship.

2. **Crafting Properties**: Write invariants and rules in CVL, starting with valid state invariants (which become trusted preconditions for other rules), then state/variable transitions, and finally high-level properties.

Project Structure

```
certora/
├── confs/                                # Prover configuration files
│   └── yield_ctf/
│       ├── valid_state.conf
│       ├── variable_transitions.conf
│       ├── state_transitions.conf
│       ├── unit_tests.conf
│       ├── high_level.conf
│       └── valid_state_mutations_depositedAmountEqualsPositions.conf
│
├── harnesses/                            # Verification harnesses
│   ├── YieldBearingConditionalTokensHarness.sol
│   └── HelperCVL.sol
│
├── mutations/                            # Mutation testing files
│   ├── add_mutation.sh                   # Helper script to create new mutations
│   └── depositedAmountEqualsPositions/   # VS-11 mutation test suite (18 mutants)
│       ├── 1.sol
│       ├── ...
│       ├── 18.sol
│       ├── mutation-testing.png
│       └── mutation-testing_patch_8.png
│
└── specs/                                # CVL specifications
    ├── setup/                            # Shared setup: ghost variables, hooks, summaries
    │   ├── yield_ctf.spec                # Main harness setup, environment constraints
    │   ├── bitops.spec                   # Bitwise operation summaries (lookup tables)
    │   ├── ct_helpers_lib.spec           # CTHelpers hash function summaries (ghost
mappings)
    │   ├── erc20.spec                    # Lightweight ERC-20 model (ghost-based)
    │   ├── venus.spec                    # Venus vToken model (fixed 1:1 exchange rate)
```

```
    │   ├── helper.spec                    # General helper definitions
    │   ├── math.spec                      # Math utility definitions
    │   ├── safe_erc20.spec                # SafeERC20 summaries
    │   └── openzeppelin/
    │       ├── erc1155.spec               # ERC-1155 internal function summaries
    │       ├── access_control.spec        # AccessControl summaries
    │       └── utils_array.spec           # Array utility summaries
    │
    └── yield_ctf/                         # Property specifications
        ├── yield_ctf_valid_state.spec     # Valid State invariants
        ├── yield_ctf_variable_transitions.spec  # Variable Transition rules
        ├── yield_ctf_state_transitions.spec     # State Transition rules
        ├── yield_ctf_unit_tests.spec      # Unit Test rules
        └── yield_ctf_high_level.spec      # High-Level rules
```

Assumptions

Formal verification requires assumptions about the code and its environment to address prover timeouts, tool limitations, and state consistency. However, incorrect assumptions can mask real bugs by excluding reachable states from analysis. To maintain transparency, all assumptions are categorized into three groups: **Safe** (real-world constraints that don't reduce security coverage), **Proved** (formally verified invariants reused as preconditions), and **Unsafe** (scope reductions necessary for tractability that may exclude valid scenarios).

Safe Assumptions

These reflect real-world constraints that don't impact security guarantees. In the codebase, every `require` statement that constitutes a safe assumption is annotated with a `"SAFE: ..."` message string.

Environment Constraints (`yield_ctf.spec`):

- The contract does not handle native ETH, so transactions carry no value
- The sender is never the zero address and never the contract itself
- Block timestamps fall within realistic bounds (up to year 2106)
- The block number is non-zero

Address Separation (`yield_ctf.spec`):

- The underlying token address is non-zero and distinct from the CTF contract
- The oracle address is non-zero and distinct from the CTF contract
- User account addresses are non-zero, distinct from the CTF contract, and distinct from each other
- The CTF contract is neither a vToken nor an underlying token
- The caller is never one of the token contracts (underlying or vToken)

ERC-20 Model (`erc20.spec`):

- Total supply equals the sum of all individual balances (standard ERC-20 solvency invariant)
- Token decimals are between 6 and 18 (realistic range for deployed tokens)
- The called token contract is never the zero address

Venus vToken Model (`venus.spec`):

- A vToken address always differs from its own underlying token address
- Different vTokens map to different underlying tokens (injectivity)
- If a token is the underlying of some vToken, it is not itself a vToken (no circular wrapping)

Proved Assumptions

These properties have been formally verified as valid state invariants and are used as trusted preconditions (via `requireInvariant`) in state transition and high-level rules. See Valid State for detailed descriptions and prover run links.

Conditional Token Invariants (`yield_ctf_valid_state.spec`):

- Condition ID zero is never prepared (VS-01)
- A prepared condition cannot have a zero oracle (VS-02)
- A prepared condition always has at least 2 outcomes (VS-03)
- The payout denominator equals the sum of payout numerators (VS-04)
- The payout numerators array length matches the outcome slot count used when creating the condition (VS-05)
- ERC-1155 token balances can only be non-zero for prepared conditions (VS-06)
- ERC-1155 positions can only exist for collateral tokens that have a registered vToken (VS-07)
- Before resolution, total supplies of both outcome positions are equal (VS-08)
- Before resolution, outstanding positions are fully backed by collateral (VS-09)
- After resolution, the weighted sum of positions is fully backed by collateral (VS-10)
- Before resolution with yield enabled, the deposited amount exactly equals outstanding positions (VS-11)
- Before resolution with yield disabled, the underlying token balance covers outstanding positions (VS-12)

Venus Integration Invariants (`yield_ctf_valid_state.spec`):

- The zero address has zero values in all underlying-related mappings (VS-13)
- Non-zero deposited amounts require the underlying to be enabled and have a registered vToken (VS-14)
- A token cannot simultaneously be registered as both an underlying and a vToken (VS-16)
- If an underlying maps to a vToken, then that vToken maps back to the same underlying (VS-17)
- If a vToken maps to an underlying and that underlying has a registered vToken, the mapping is consistent (VS-18)
- Deposited amounts are fully backed by the corresponding vToken value (VS-19)

> Note: VS-11 (`depositedAmountEqualsPositions`) uses strict equality (`==`) rather than `>=`, even though a weaker `>=` would be sound — a donate-then-enable flow causes `enableUnderlying` to deposit the entire contract balance (including donations) into Venus, inflating `depositedAmount` above outstanding positions. The strict equality is intentional: it catches any discrepancy, no matter how small, making the invariant maximally sensitive. To handle the `enableUnderlying` case, a dedicated `preserved` block assumes no untracked balance exists before enabling.

> Note: VS-15 (`enabledUnderlyingHasVToken`) is violated in `enableUnderlying()` — a non-connected token can be enabled. This has no meaningful security impact and is not used as a `requireInvariant`.

Unsafe Assumptions

These reduce verification scope to make the problem tractable for the prover. In the codebase, every `require` statement that constitutes an unsafe assumption is annotated with an `"UNSAFE: ..."` message string.

Scope Restrictions (`yield_ctf.spec`):
Bounding the number of tokens, accounts, conditions, and outcomes serves two purposes: it prevents prover timeouts on unbounded quantifiers, and it makes it possible to reason about total supply as a concrete sum of all user balances — enabling protocol-wide invariants such as "the deposited amount exactly equals outstanding positions."

- Only a single underlying collateral token is supported
- Only a single oracle address is used for condition creation
- Only a single question identifier is used
- The maximum outcome slot count is capped at 2 (binary markets only)
- ERC-1155 balances are tracked for at most 2 user accounts
- Only a single condition can be in a prepared state at a time
- Only initial (non-nested) positions are verified — parent collection is always zero
- Conditions, collateral tokens, and accounts outside the supported set are assumed to hold zero ERC-1155 balances
- Payout numerators do not exist for unsupported conditions
- Payout numerators array length never exceeds the maximum supported outcome slot count

Bit Operations and Index Sets (`bitops.spec`):
By default the Certora Prover overapproximates bitwise operations rather than building precise models, which produces spurious counterexamples or causes the prover to crash. To work around this, all bitwise operations in the Solidity source were extracted into separate internal functions and replaced with CVL summaries. Every such code change is marked with an `ORIGINAL:` comment showing the original expression. Inputs are bounded to the valid range for 2-outcome conditions to keep the CVL replacements sound.

- Index sets and free index sets are restricted to values at most 3 (valid for 2-outcome conditions)
- Bit position access is limited to indices below the maximum supported outcome slot count

Condition and Collection Helpers (`ct_helpers_lib.spec`):
The CTF helper library uses hash-based lookups (condition IDs, collection IDs) that are opaque to the prover. Restricting inputs to a single supported condition and zero parent collection makes these lookups deterministic and allows the prover to relate positions back to their underlying condition.

- Only a single condition with supported parameters (oracle, question ID, outcome slot count) is used
- Collections are derived only from the zero parent (no nested conditional positions)

Amount Bounds — Overflow Prevention (`erc20.spec`, `erc1155.spec`, `yield_ctf.spec`):
Ghost variables use `mathint` (unbounded integers), so without explicit bounds the prover must consider astronomically large values that cannot occur in practice. Capping amounts well below `uint256` eliminates spurious overflow counterexamples while still covering all realistic token supplies.

- ERC-20 balances are bounded by max uint128 to prevent arithmetic overflows
- ERC-20 allowances are bounded by max uint128 to prevent arithmetic overflows

- ERC-1155 balances are bounded by max uint112 to prevent arithmetic overflows
- Deposited amounts are bounded by max uint128 to prevent arithmetic overflows

Account Restrictions (`erc20.spec`, `yield_ctf.spec`):
Restricting accounts to a finite set allows total supply to be expressed as a concrete sum and ensures storage hooks only fire for known addresses, preventing the prover from reasoning about an unbounded address space.

- All ERC-20 operations restrict accounts to a predefined set of up to 5 addresses per token
- ERC-1155 storage hooks for balances and operator approvals restrict accounts to the 2 supported addresses

Venus Exchange Rate (`venus.spec`):
The Venus protocol's variable exchange rate introduces non-linear arithmetic that is intractable for the SMT solver. Fixing it at 1:1 makes vToken amounts equal to underlying amounts, reducing the problem to linear arithmetic while preserving the deposit/redeem control flow.

- The vToken exchange rate is fixed at 1:1 — minting, redeeming, and balance-of-underlying all use a 1e18 exchange rate

Prover Configuration:

- Loop unrolling is capped at 2 iterations across all configurations

Verification Properties

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

- ✅ Verified successfully
- ⚠️ Timeout
- ❌ Violated (indicates a potential issue)

Valid State

Valid State properties define the fundamental invariants that must always hold true throughout the protocol's lifecycle. These are parametric — each property is automatically verified against every external function in the contract, including functions added in the future.

Conditional Token Invariants

Property	Name	Description	Status	Notes
VS-01	`conditionIdZeroNotPrepared`	ConditionId 0 is never prepared	✅	
VS-02	`preparedConditionNotWithZeroOracle`	Prepared condition cannot be created by zero oracle	✅	
VS-03	`preparedConditionHasMinTwoOutcomes`	Prepared condition has at least 2 outcomes	✅	
VS-04	`payoutDenominatorEqualsSumNumerators`	Payout denominator equals sum of payout numerators	✅	
VS-05	`conditionIdMatchesOutcomeSlotCount`	Prepared conditionId length matches outcomeSlotCount in hash	✅	
VS-06	`erc1155ExistsImpliesConditionPrepared`	ERC-1155 tokens exist only for prepared conditions	✅	
VS-07	`erc1155RequiresRegisteredVToken`	ERC-1155 positions exist only for collateral with registered vToken	✅	
VS-08	`positionSuppliesEqualBeforeResolution`	Total supplies of both outcomes are equal before resolution	✅	
VS-09	`erc1155PositionsFullyBackedUnresolved`	ERC-1155 positions total supply backed by collateral (before resolution)	✅	
VS-10	`erc1155PositionsFullyBackedResolved`	ERC-1155 positions total supply backed by collateral (after resolution)	✅	✅ `redeemPositions()` passed ✅ `splitPosition()` passed ⚠️ `mergePositions()` timeout

Property	Name	Description	Status	Notes
VS-11	`depositedAmountEqualsPositions`	Before resolution, deposited amount tracks outstanding positions exactly	✅	Mutation tested — INFO: `preserved` block for `enableUnderlying` requires unsafe assumption (balance == outstanding positions)
VS-12	`underlyingBalanceBacksPositions`	Before resolution with yield disabled, underlying balance covers positions	✅	

Venus Integration Invariants

Property	Name	Description	Status	Notes
VS-13	`zeroUnderlyingHasZeroValues`	Zero address underlying has zero values in all mappings	✅	
VS-14	`depositsRequireEnabledAndVToken`	Deposits require underlying to be enabled and have a vToken	✅	
VS-15	`enabledUnderlyingHasVToken`	Enabled underlying must have a vToken	❌	Violated in `enableUnderlying()`: non-connected token can be enabled — no meaningful impact
VS-16	`underlyingNotVToken`	Token cannot be both underlying and vToken simultaneously	✅	
VS-17	`vTokenMappingConsistency`	If underlying maps to vToken, then vToken maps back to underlying	✅	
VS-18	`vTokenReverseMappingConsistency`	If vToken maps to underlying AND underlying is registered, mapping is consistent	✅	
VS-19	`depositsFullyBacked`	Deposited amount must be fully backed by vToken value	✅	

Variable Transitions

Variable Transition properties verify that specific storage variables change only under expected conditions (monotonicity — set once and never changed). These are parametric — each property is automatically verified against every external function in the contract, including functions added in the future.

Property	Name	Description	Status	Notes
VT-01	`transitionUnderlyingToVTokenSetOnce`	underlyingToVToken can only be set once (from zero to non-zero)	✅	

Property	Name	Description	Status	Notes
VT-02	`transitionPayoutDenominatorSetOnce`	payoutDenominator can only be set once (from zero to non-zero)	✅	
VT-03	`transitionPayoutNumeratorsSetOnce`	payoutNumerators can only be set once (from zero to non-zero)	✅	
VT-04	`transitionPayoutNumeratorsLengthSetOnce`	payoutNumerators length can only be set once (from zero to non-zero)	✅	

State Transitions

State Transition properties verify the correctness of transitions between valid states. These are parametric — each property is automatically verified against every external function in the contract, including functions added in the future.

Venus Deposit/Withdrawal Correlation

Property	Name	Description	Status	Notes
ST-01	`depositedAmountIncreasesTogetherWithVTokenBalance`	depositedAmount increase must correlate with vToken balance increase	✅	
ST-02	`depositedAmountDecreasesTogetherWithVTokenBalance`	depositedAmount decrease must correlate with vToken balance decrease	✅	
ST-03	`vTokenBalanceDecreaseOnlyViaBurnOnRedeem`	CTF vToken balance can only decrease via redeem	✅	

ERC-1155 Transfer Isolation

Property	Name	Description	Status	Notes
ST-04	`erc1155TransferDoesNotAffectTotalBalance`	ERC-1155 transfer does not affect total balance	✅	
ST-05	`erc1155TransferDoesNotAffectVTokenBalance`	ERC-1155 transfer does not affect vToken balance on CTF	✅	
ST-06	`erc1155TransferDoesNotAffectUnderlyingBalance`	ERC-1155 transfer does not affect underlying balance on CTF	✅	
ST-07	`erc1155TransferDoesNotAffectDepositedAmount`	ERC-1155 transfer does not affect depositedAmount	✅	

Minting, Access Control, and Resolution

Property	Name	Description	Status	Notes
ST-08	`erc1155MintRequiresBurnAnotherOrCollateralIn`	ERC-1155 mint requires either another position burn or collateral transfer in	✅	
ST-09	`erc1155TransferRequiresWhitelistedParticipant`	ERC-1155 transfer with transferControl requires whitelisted participant	✅	
ST-10	`reportPayoutsSetsAtLeastOneNumerator`	reportPayouts must set at least one numerator when setting denominator	✅	

Property	Name	Description	Status	Notes
ST-11	`underlyingDecreaseRequiresYieldManagerRole`	Underlying balance decrease requires YIELD_MANAGER_ROLE when positions unchanged	✅	

High-Level

High-Level properties verify multi-step business logic, user isolation across core operations, and monotonicity of core operations.

Property	Name	Description	Status	Notes
HL-01	`splitMergePreservesCollateral`	Split + Merge keeps collateral unchanged	✅	
HL-02	`splitThenMergePositionsRestored`	Split + Merge preserves ERC-1155 position balances	✅	
HL-03	`splitDoesNotAffectOtherUsersPositions`	Split does not affect other users' ERC-1155 positions	✅	
HL-04	`mergeDoesNotAffectOtherUsersPositions`	Merge does not affect other users' ERC-1155 positions	✅	
HL-05	`redeemDoesNotAffectOtherUsers`	Redeem does not affect other users' positions	✅	
HL-06	`redeemBurnsCallerTokens`	Redeem does not increase caller's position balance	✅	
HL-07	`redeemDoesNotIncreaseCollateral`	Redeem does not increase collateral backing	✅	

Unit Tests

Unit Test properties verify basic behavior of individual functions: revert conditions, direct effects, and non-effects on unrelated state.

Revert Checks

Property	Name	Description	Status	Notes
UT-01	`reportPayoutsImmutability`	reportPayouts reverts for already resolved condition	✅	
UT-02	`prepareConditionIdempotencyGuard`	prepareCondition reverts for already prepared condition	✅	
UT-03	`redeemRevertsIfNotResolved`	redeemPositions reverts if condition not resolved	✅	
UT-04	`splitRevertsWhenGatedAndUnauthorized`	splitPosition reverts when gated and caller lacks role	✅	
UT-05	`mergeRevertsWhenGatedAndUnauthorized`	mergePositions reverts when gated and caller lacks role	✅	

Property	Name	Description	Status	Notes
UT-06	`claimYieldOnlyYieldManager`	claimYield reverts when caller lacks YIELD_MANAGER_ROLE	✅	

Direct Effects

Property	Name	Description	Status	Notes
UT-07	`reportPayoutsSetsResolved`	reportPayouts sets condition as resolved	✅	
UT-08	`prepareConditionSetsOutcomeSlots`	prepareCondition initializes outcome slots	✅	

Non-Effects

Property	Name	Description	Status	Notes
UT-09	`reportPayoutsDoesNotChangeCollateral`	reportPayouts does not change collateral backing	✅	
UT-10	`reportPayoutsDoesNotAffectPositions`	reportPayouts does not affect ERC-1155 balances	✅	

Mutation Testing

Valid state invariants are powerful — a single well-crafted invariant can catch a wide range of bugs across multiple functions. To demonstrate this, we use `depositedAmountEqualsPositions` (VS-11) as a case study, showing through mutation testing that this one invariant detects 18 distinct bugs in `splitPosition` and `mergePositions`.

What is Mutation Testing

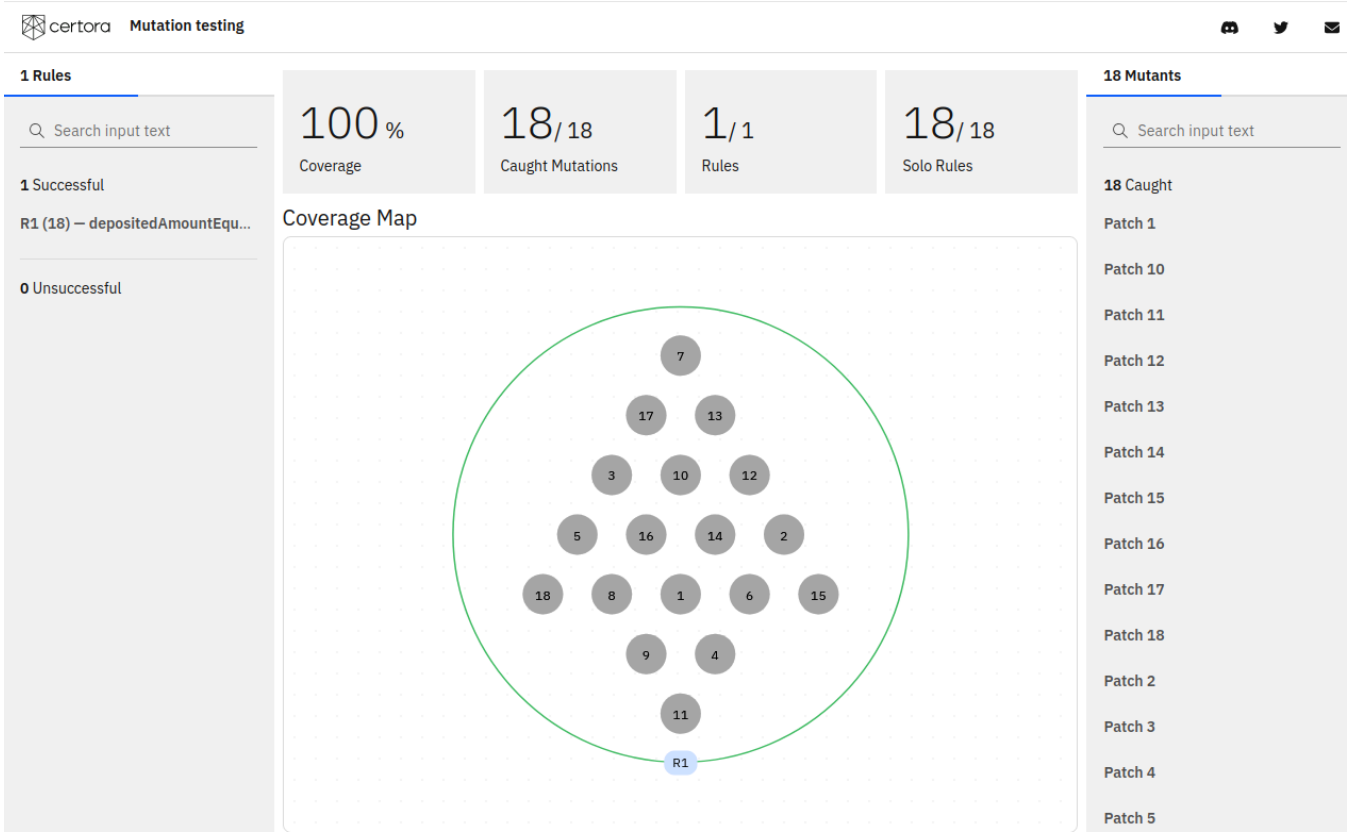
Mutation testing validates the strength of a formal specification by injecting small syntactic changes (mutations) into the source code and verifying that the spec catches them. Each mutation represents a potential bug — a removed function call, a flipped condition, a swapped parameter.

The certoraMutate tool automates this process: it applies each mutant to the source, runs the prover against the specified rule, and reports whether the invariant caught the introduced defect.

VS-11: depositedAmountEqualsPositions

The `depositedAmountEqualsPositions` invariant asserts that when yield is enabled and the condition is unresolved, the Venus deposited amount exactly equals the total outstanding ERC-1155 positions. This strict equality makes the invariant maximally sensitive to any accounting discrepancy.

Result: **18/18 mutations caught ❌** — the invariant catches every injected bug.



splitPosition mutations

#1 ❌ Remove Venus deposit after collateral transfer.

```
 if (underlyingIsEnabled[address(collateralToken)]) {
-    _mintVToken(address(collateralToken), amount);
+    // removed
 }
```

#2 ❌ Flip full-partition check from `==` to `!=`.

```
-if (freeIndexSet == 0) {
+if (freeIndexSet != 0) {
```

#3 ❌ Flip parent collection check from `==` to `!=`.

```
-if (parentCollectionId == bytes32(0)) {
+if (parentCollectionId != bytes32(0)) {
```

#4 ❌ Remove entire collateral handling block (transfer, Venus deposit, and partial-split burn).

```
-if (freeIndexSet == 0) {
-    if (parentCollectionId == bytes32(0)) {
-        require(collateralToken.transferFrom(msg.sender, address(this), amount), ...);
-        if (underlyingIsEnabled[address(collateralToken)]) {
-            _mintVToken(address(collateralToken), amount);
-        }
-    } else {
-        _burn(msg.sender, CTHelpers.getPositionId(collateralToken, parentCollectionId),
amount);
-    }
-} else {
-    _burn(msg.sender, CTHelpers.getPositionId(collateralToken, ...), amount);
-}
+// removed
```

#5 ❌ Remove position minting call.

```
-_mintBatch(msg.sender, positionIds, amounts, "");
+// removed
```

#11 ❌ Negate yield-enabled check before Venus deposit.

```
-if (underlyingIsEnabled[address(collateralToken)]) {
+if (!underlyingIsEnabled[address(collateralToken)]) {
```

#12 ❌ Off-by-one in Venus deposit amount.

```
-_mintVToken(address(collateralToken), amount);
+_mintVToken(address(collateralToken), amount - 1);
```

#13 ❌ Zero out mint amounts for all positions.

```
-amounts[i] = amount;
+amounts[i] = 0;
```

mergePositions mutations

#6 ❌ Skip Venus redeem, transfer raw amount directly.

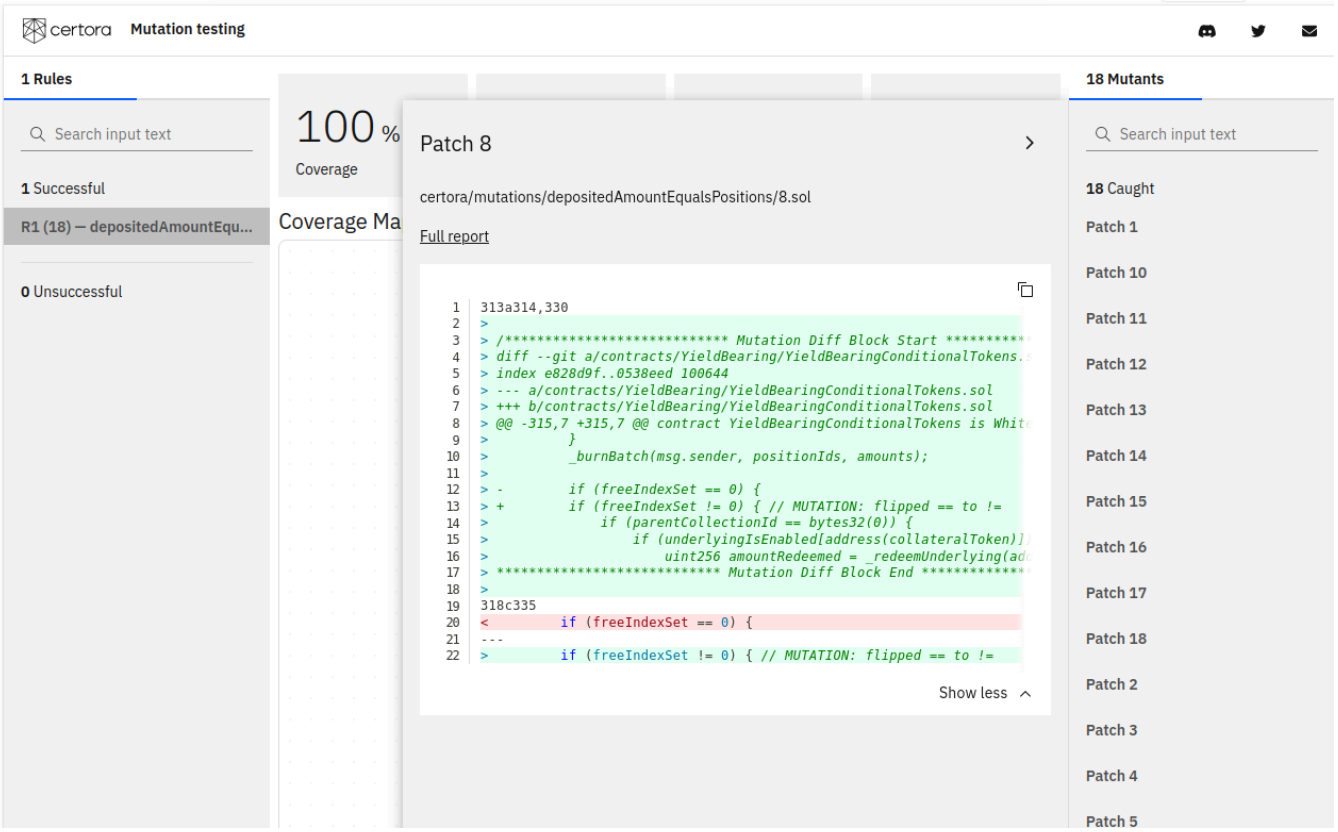
```
 if (underlyingIsEnabled[address(collateralToken)]) {
-    uint256 amountRedeemed = _redeemUnderlying(address(collateralToken), amount);
-    require(collateralToken.transfer(msg.sender, amountRedeemed), ...);
+    require(collateralToken.transfer(msg.sender, amount), ...);
 }
```

#7 ❌ Remove position burn batch call.

```
-_burnBatch(msg.sender, positionIds, amounts);
+// removed
```

#8 ❌ Flip full-partition check from `==` to `!=`.

```
-if (freeIndexSet == 0) {
+if (freeIndexSet != 0) {
```



#9 ❌ Negate yield-enabled check before Venus redeem.

```
-if (underlyingIsEnabled[address(collateralToken)]) {
+if (!underlyingIsEnabled[address(collateralToken)]) {
```

#10 ❌ Flip parent collection check from `==` to `!=`.

```
-if (parentCollectionId == bytes32(0)) {
+if (parentCollectionId != bytes32(0)) {
```

#14 ❌ Off-by-one over-redeem from Venus.

```
-uint256 amountRedeemed = _redeemUnderlying(address(collateralToken), amount);
+uint256 amountRedeemed = _redeemUnderlying(address(collateralToken), amount + 1);
```

#15 ❌ Zero Venus redeem amount.

```
-uint256 amountRedeemed = _redeemUnderlying(address(collateralToken), amount);
+uint256 amountRedeemed = _redeemUnderlying(address(collateralToken), 0);
```

#16 ❌ Zero out burn amounts for all positions.

```
-amounts[i] = amount;
+amounts[i] = 0;
```

#17 ❌ Double burn amounts for all positions.

```
-amounts[i] = amount;
+amounts[i] = amount * 2;
```

#18 ❌ Wrong mapping key in yield-enabled check.

```
-if (underlyingIsEnabled[address(collateralToken)]) {
+if (underlyingIsEnabled[address(this)]) {
```

Setup and Execution

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

Common Setup (Steps 1–4)

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

1. Install Java (tested with JDK 21)

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

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

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

3. Install Certora CLI. To match a specific prover version, pin it explicitly (e.g. `certora-cli==8.6.3`)

```
pipx install certora-cli
```

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

```
pipx install solc-select
solc-select install 0.8.24
solc-select use 0.8.24
```

Remote Execution

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

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

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

Local Execution

Follow the full build instructions in the CertoraProver repository (v8.6.3). Once the local prover is installed it takes priority over the remote cloud by default. Tested on Ubuntu 24.04.

1. Install prerequisites

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

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

# LLVM tools
sudo apt install llvm

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

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

2. Set up build output directory

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

3. Clone and build

```
git clone --recurse-submodules https://github.com/Certora/CertoraProver.git
cd CertoraProver
git checkout tags/8.6.3
./gradlew assemble
```

4. Verify installation with test example

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

Running Verification

Quick Runs

`variable_transitions` and `unit_tests` are lightweight and can be run as full conf files:

```
certoraRun certora/confs/yield_ctf/variable_transitions.conf
certoraRun certora/confs/yield_ctf/unit_tests.conf
```

> **Note:** `valid_state`, `state_transitions`, and `high_level` may time out when running all rules at once. Run them per-rule using `--rule`, `--method`, and `--exclude method` as shown below.

Valid State

```
certoraRun certora/confs/yield_ctf/valid_state.conf --rule conditionIdZeroNotPrepared
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
preparedConditionNotWithZeroOracle
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
preparedConditionHasMinTwoOutcomes
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
payoutDenominatorEqualsSumNumerators
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
conditionIdMatchesOutcomeSlotCount
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155ExistsImpliesConditionPrepared
certoraRun certora/confs/yield_ctf/valid_state.conf --rule erc1155RequiresRegisteredVToken
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
positionSuppliesEqualBeforeResolution
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155PositionsFullyBackedUnresolved
certoraRun certora/confs/yield_ctf/valid_state.conf --rule depositedAmountEqualsPositions
certoraRun certora/confs/yield_ctf/valid_state.conf --rule underlyingBalanceBacksPositions
certoraRun certora/confs/yield_ctf/valid_state.conf --rule zeroUnderlyingHasZeroValues
certoraRun certora/confs/yield_ctf/valid_state.conf --rule depositsRequireEnabledAndVToken
certoraRun certora/confs/yield_ctf/valid_state.conf --rule enabledUnderlyingHasVToken
certoraRun certora/confs/yield_ctf/valid_state.conf --rule underlyingNotVToken
certoraRun certora/confs/yield_ctf/valid_state.conf --rule vTokenMappingConsistency
certoraRun certora/confs/yield_ctf/valid_state.conf --rule vTokenReverseMappingConsistency
certoraRun certora/confs/yield_ctf/valid_state.conf --rule depositsFullyBacked
```

`erc1155PositionsFullyBackedResolved` requires per-method splits to avoid timeouts:

```
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155PositionsFullyBackedResolved --method
"splitPosition(address,bytes32,bytes32,uint256[],uint256)"
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155PositionsFullyBackedResolved --method
"mergePositions(address,bytes32,bytes32,uint256[],uint256)"
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155PositionsFullyBackedResolved --method
"redeemPositions(address,bytes32,bytes32,uint256[])"
certoraRun certora/confs/yield_ctf/valid_state.conf --rule
erc1155PositionsFullyBackedResolved --exclude_method
"splitPosition(address,bytes32,bytes32,uint256[],uint256)" --exclude_method
"mergePositions(address,bytes32,bytes32,uint256[],uint256)" --exclude_method
"redeemPositions(address,bytes32,bytes32,uint256[])"
```

State Transitions

```
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
depositedAmountIncreasesTogetherWithVTokenBalance
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
depositedAmountDecreasesTogetherWithVTokenBalance
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
vTokenBalanceDecreaseOnlyViaBurnOnRedeem
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155TransferDoesNotAffectTotalBalance
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155TransferDoesNotAffectVTokenBalance
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155TransferDoesNotAffectUnderlyingBalance
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155TransferDoesNotAffectDepositedAmount
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155MintRequiresBurnAnotherOrCollateralIn
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
erc1155TransferRequiresWhitelistedParticipant
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
reportPayoutsSetsAtLeastOneNumerator
certoraRun certora/confs/yield_ctf/state_transitions.conf --rule
underlyingDecreaseRequiresYieldManagerRole
```

High-Level

```
certoraRun certora/confs/yield_ctf/high_level.conf --rule splitMergePreservesCollateral
certoraRun certora/confs/yield_ctf/high_level.conf --rule splitThenMergePositionsRestored
certoraRun certora/confs/yield_ctf/high_level.conf --rule
splitDoesNotAffectOtherUsersPositions
certoraRun certora/confs/yield_ctf/high_level.conf --rule
mergeDoesNotAffectOtherUsersPositions
certoraRun certora/confs/yield_ctf/high_level.conf --rule redeemDoesNotAffectOtherUsers
certoraRun certora/confs/yield_ctf/high_level.conf --rule redeemBurnsCallerTokens
certoraRun certora/confs/yield_ctf/high_level.conf --rule redeemDoesNotIncreaseCollateral
```

Running Mutation Testing

Run the mutation test suite for `depositedAmountEqualsPositions` (VS-11):

```
certoraMutate
certora/confs/yield_ctf/valid_state_mutations_depositedAmountEqualsPositions.conf
```

Creating Mutations for Other Invariants

1. **Create the mutations directory**: `mkdir -p certora/mutations/<invariantName>`
2. **Introduce a mutation** into the source contract (e.g., comment out a line, flip a condition)
3. **Run the helper script** to snapshot the mutation:

   ```
   ./certora/mutations/add_mutation.sh <invariantName>
   contracts/YieldBearing/YieldBearingConditionalTokens.sol
   ```

   This auto-numbers the mutation file (e.g., `1.sol`, `2.sol`, ...) and embeds the diff block. The original source is automatically restored via `git restore`.
4. **Create a conf file** (copy an existing one and add the `mutations` section pointing to your new directory)
5. **Run** `certoraMutate` with your new conf file

Source Code Modifications

The Certora Prover overapproximates bitwise operations by default, which produces spurious counterexamples or causes the prover to crash. To work around this, all inline bitwise expressions in `splitPosition`, `mergePositions`, and `redeemPositions` were extracted into internal helper functions. These functions are then replaced with precise CVL summaries defined in `bitops.spec`.

Each replacement site in the Solidity source is marked with an `// ORIGINAL:` comment showing the original expression, making it straightforward to audit the modifications.

Helper Functions

The following internal functions were added to the `BitOps Certora Internal Functions` block in `YieldBearingConditionalTokens.sol`. Extracting bitwise expressions into separate functions allows them to be summarized with precise CVL implementations in `bitops.spec`.

Replacement Sites

All 10 replacement sites across the 3 functions:

`splitPosition`

```
// ORIGINAL: uint fullIndexSet = (1 << outcomeSlotCount) - 1;
uint fullIndexSet = _createFullIndexSet(outcomeSlotCount);

// ORIGINAL: require((indexSet & freeIndexSet) == indexSet, "partition not disjoint");
require(_isSubset(indexSet, freeIndexSet), "partition not disjoint");

// ORIGINAL: freeIndexSet ^= indexSet;
freeIndexSet = _removeBits(freeIndexSet, indexSet);

// ORIGINAL: CTHelpers.getCollectionId(parentCollectionId, conditionId, fullIndexSet ^
freeIndexSet)
CTHelpers.getCollectionId(parentCollectionId, conditionId, _complement(fullIndexSet,
freeIndexSet))
```

`mergePositions`

```
// ORIGINAL: uint fullIndexSet = (1 << outcomeSlotCount) - 1;
uint fullIndexSet = _createFullIndexSet(outcomeSlotCount);

// ORIGINAL: require((indexSet & freeIndexSet) == indexSet, "partition not disjoint");
require(_isSubset(indexSet, freeIndexSet), "partition not disjoint");

// ORIGINAL: freeIndexSet ^= indexSet;
freeIndexSet = _removeBits(freeIndexSet, indexSet);

// ORIGINAL: CTHelpers.getCollectionId(parentCollectionId, conditionId, fullIndexSet ^
freeIndexSet)
CTHelpers.getCollectionId(parentCollectionId, conditionId, _complement(fullIndexSet,
freeIndexSet))
```

`redeemPositions`

```
// ORIGINAL: uint fullIndexSet = (1 << outcomeSlotCount) - 1;
uint fullIndexSet = _createFullIndexSet(outcomeSlotCount);

// ORIGINAL: if (indexSet & (1 << j) != 0) {
if (_isBitSet(indexSet, j)) {
```