report.md

 Formal Verification of Balancer Stable Pool (June - September 2022)

Summary

This document describes the specification and verification of Balancer's contracts using the Certora Prover. The work was undertaken from June 21st to September 23rd. The latest commit that was reviewed and run through the Certora Prover was commit `af9e9eb`

The scope of our verification was the following contracts and their various components:

- [`ComposableStablePool.sol`]
- [`StablePool.sol`]
- [`WordCodec.sol`]

The Certora Prover proved the implementation of the Balancer contracts is correct with respect to the formal rules written by the Balancer and the Certora teams. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. Additionally the code was manually reviewed by two researchers from Certora.

List of Main Issues Discovered

Severity: Low

Issue:	disableRecoveryMode() can be called while not in recovery mode
Severity:	LOW
Rules Broken:	None

Issue:	disableRecoveryMode() can be called while not in recovery mode
Description:	When disableRecoveryMode() is called, it does not check whether or not the system is currently under recovery mode. Rather, it sets recovery mode to false and conducts subsequent updates.
Response:	In that case the system behaves as if recovery mode was enabled and immediately disabled. It's an interesting finding, though it does not changes the permissions model much, since disabling recovery mode is an extremely rare occurrence It does make for a weird footnote
--------	--------
Issue:	exiting in recovery mode is advantageous to users
Severity:	INFORMATIONAL
Rules Broken:	None
Description:	Exiting in recovery mode is sometimes more advantageous over exiting in non-recovery mode. We have investigated further to see if a join/exit sequence can be profitable or be used to manipulate prices, however we have not found a scenario for intentional manipulation
Response:	Aware of potentially advantageous exits in recovery mode.
--------	--------
Issue:	Denial of Service for increasing amplification factor
Severity:	INFORMATIONAL
Rules Broken:	amplificationUpdateCanFinish
Description:	The startAmplificationUpdate() function allows for value of endTime, the result is that the updating value will continue to be true stopping the system from ever allowing for more amplificationFactorUpdates
Response:	This issue can be ignored by using the stopAmplificationParameterUpdate() function

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

✔ indicates the rule is formally verified on the latest reviewed commit. We write ✔* when the rule was verified on a simplified version of the code (or under some assumptions).

✘ indicates the rule was violated under one of the tested versions of the code.

✍ indicates the rule has not been checked on the current version.

⏲ indicates that some functions cannot be verified because the rules timed out

Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

Verification of ComposableStablePool.sol

Assumptions and Simplifications

`ComposableStablePool` contract handles up to 6 tokens including its native BPT token. However, in order to avoid prover timing out, we need to limit totalTokens and loop iterations to 3, and, at times, even to 2, while fixing the non-bpt token to 1.

We assume all non-view functions are only to be called by the vault, and proper implementation of the vault. Such as the ordering of balances and number of tokens provided being correct.

For rules regarding the amplification factor we make the following assumptions about the state variables of the system:

- The minimum update time is between 0 and 1 day (`_MIN_UPDATE_TIME`)
- The amplification factor may not increase or decrease by a factor of 2 over a two day period (`_MAX_AMP_UPDATE_DAILY_RATE`)
- The minimum amplification factor is greater than 0 (`_MIN_AMP`)
- The maximum amplification factor is between the minimum amplification factor and 100000 (normally this is set to 5000) (`_MAX_AMP`) Note that many of these variables are hard coded as immutable constants, however the tool will assume any possible values for those constants unless otherwise constrained.

For the purpose of simplification, we replaced the StableMath Library with the contract `StableMathHarness.sol` . The following functions have been replaced to return arbitrary, but consistent values based on the parameters, within the range of possible values for the originals of each given function: `* _calcOutGivenIn() * _calcInGivenOut() * calcInGivenOut() * _getTokenBalanceGivenInvariantAndAllOtherBalances`

The following functions have been set to return nondeterministic values for the sake of simplification: `getAuthorizor()` (the address is nondet but does not affect assignment of authorizor functions), `registerPool()` , `registerTokens()` , `getPoolTokenInfo()` , `getPoolTokens()` , `getProtocolFeesCollector()` , `_upscaleArray()` , `_downscaleUp()` , `_downscaleUpArray()` , `_downscaleDownArray()` , `_joinExactTokensInForBPTOut()` , `_joinTokenInForExactBPTOut()` , `_exitBPTInForExactTokensOut()` , `_exitExactBPTInForTokenOut()` , `_updateInvariantAfterJoinExit()` , `getRate()` ,

Harnessing

We harnessed StableMath functions such as `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` to return arbitrary but deterministic value given fixed inputs. We also harnessed certain `ComposableStablePool` functions to expose certain variables that are otherwise inaccessible or difficult to access.

Munging

To avoid timeouts, we munged certain functions in `ComposableStablePool` . For example, `_getGrowthInvariants` may calculate invariants in different ways given specified inputs, balances or adjustedBalances. However, loops in `_getAdjustedBalances` will timeout in most circumstances. We, therefore, munged the inputs to all use balances instead of `adjustedBalances` . We, then, prove the equivalence of balances and `adjustedBalances` in relevant conditions.

Properties

(✔) invariant `basicOperationsRevertOnPause` : All basic operations must revert while in a paused state. This assumes the system is not in recovery mode initially.

(✔) rule `pauseStartOnlyPauseWindow` : If a function sets the contract into pause mode, it must only be during the `pauseWindow` . This assumes the system is not in paused state initially.

(✔) rule `unpausedAfterBuffer` : After the buffer window finishes, the contract may not enter the paused state.

(✔) rule `prOtherFunctionsAlwaysRevert` : If both paused and recovery mode is active, the set functions must always revert. This assumes the system is in paused and recovery mode initially.

(✔) rule `recoveryExitNoStableMath` : In recovery mode, exit never calls any simple math functions. This assumes `isRecoveryModeExitKind` returns true.

(✔) rule `recoveryExitNoExternalCalls` : In recovery mode, exitPool must not call any external contracts. This assumes `isRecoveryModeExitKind` returns true.

(✔) rule `ZeroOwnerPercentageInRecovery : _getProtocolPoolOwnershipPercentage` must always return 0 if recovery mode is enabled. This assumes the system is in recovery mode initially.

(✔) rule `ZeroOwnerPercentageAfterDisablingRecovery : disableRecoveryMode` should not change `virtualSupply` . Immediately after disabling, `_getProtocolPoolOwnershipPercentage` should return 0 fee percentage. This assumes: * All invariants returned by `getGrowthInvariant` are the same, which is proved by next rule. * There are only 3 tokens in total. * If `_isTokenExemptFromYieldProtocolFee` is true for a token, then the token must have `rateProvider` set for it.

(✔) rule `DisableRecoveryModeChangesStates : disableRecoveryMode` should update `lastJoinExitAmp` , `lastPostJoinExitInvariant` , as well as rate cache if `rateProvider` has been set, so that balance always equals adjusted balance. This assumes: * There are only two tokens in total, with the 2nd token being bpt. * If `_isTokenExemptFromYieldProtocolFee` is true for a token, then the token must have `rateProvider` set for it. * to avoid timeout, instead of using the existing balance in the system, we set the balance to an arbitrage value before calculating the adjusted balance.

(✔) rule `onlyOnJoinPoolCanAndMustInitialize : totalSupply` must be non-zero if and only if `onJoinPool` is successfully called. Additionally, the balance of the zero address must be non-zero if `onJoinPool` was successfully called. This assumes the system starts with 0 `totalSupply` .

(✔) rule `cantBurnZerosBPT` : The zero address's BPT balance must never go from nonzero to zero. We assume no one has access to the private keys of the zero address.

(✍) rule `noFreeMinting` : The total supply of BPT must not increase if the total tokens held by the pool don't increase. * We were unable to prove this rule on ComposableStablePool due to the increased number of minimum tokens in the pool.

(![FAILING]) rule `amplificationFactorBounded` : The amplification factor must not go past the set minimum amp and maximum amp * This rule passes on the preserve, however it fails the constructor.

(✔) rule `amplificationFactorFollowsEndTime` : After starting an amplification factor increase and calling an arbitrary function, for some e later than initial increase amplification factor must be less than or equal value set. We split this rule into two cases, amplification factor is increasing and it's decreasing, for the sake of timeouts.

(✔) rule `amplificationFactorNoMoreThanDouble` : The amplification factor may not increase by more than a factor of two in a given day. This rule has been split into two cases, increasing and decreasing, for the sake of handling timeouts.

(✔) rule `amplificationFactorUpdatingOneDay` : If the amplification factor starts updating, then it must continue so for one day.

(✘) rule `amplificationUpdateCanFinish` : If you start an amplification update, it must be able to finish within a large number of days. We choose 1000 days as our metric for an excessive amount of days. This assumes the system is not in update mode initially.

(✔) rule `noDoubleUpdate` : You must not be able to change the amplification factor if it is currently in the process of updating.

(✔) rule `ampStoreAndReturn` : Storing a value with `_setAmplificationData` must always return the set value through `getAmplificationFactor` .

Verification of StablePool.sol

- This document describes the specification and verification of Balancer's `StablePool` contract using the Certora Prover. The work was undertaken from June 2022 to September 2022. The latest commit that was reviewed and run through the Certora Prover was commit 0977c144.
  The scope of our verification was `StablePool.sol` and its dependencies.

Due to the complexity of the code, which involves heavy use of nonlinear arithmetic, the Certora prover is unable to prove many complex functions, thus limiting the scope to pause and recovery mode and the most important high-level rules.

Assumptions and Simplifications

We assume all non-view functions are only to be called by the vault, and proper implementation of the vault. Such as the ordering of balances, number of tokens provided being correct, and all tokens being unique.

For rules regarding the amplification factor we make the following assumptions about the state variables of the system:

- The minimum update time is between 0 and 1 day (`_MIN_UPDATE_TIME`)
- The amplification factor may not increase or decrease by a factor of 2 over a two day period (`_MAX_AMP_UPDATE_DAILY_RATE`)
- The minimum amplification factor is greater than 0 (`_MIN_AMP`)
- The maximum amplification factor is between the minimum amplification factor and 100000 (normally this is set to 5000) (`_MAX_AMP`) Note that many of these variables are hard coded as immutable constants, however the tool will assume any possible values for those constants unless otherwise constrained.

Additionally, we make assumptions about complex functions in order to simplify the code and make it generating proofs and counterexamples manageable. We assume that `StablePool` has 2 tokens because further loop unrolling would cause timeouts. We have found that issues that exist in pools with 3+ tokens also exist in pools with 2 tokens.

We summarize `_calculateInvariant` by returning an arbitrary value which is greater than or equal to the sum of the 2 token balances and less than or equal to the product. We also assume the value returned is deterministic. We also summarize `_getTokenBalanceGivenInvariantAndAllOtherBalances` to always return an increased balance if the invariant increased. This summarization is only applied in the case when a pool is being joined using `onJoinPool` , specifically in the `noFreeMinting` rules.

Harnessing

The contract `StablePoolHarness` inherits from `StablePool` . In the harness we added some functionality to `onJoinPool` and `onExitPool` to track token transfers which would normally be executed by the vault. Additionally, we implemented some extra functions to more easily access info and make requirements about dynamic arrays.

Munging

We munged `StableMath` functions `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` in order to be able to summarize them. Due to limitations of the CVL, we can not summarize functions with dynamic array inputs to CVL functions so these two functions were munged to call a secondary function with all the same variables except for `balances` which was replaced by `balance1` and `balance2`. We also munged `_mutateAmounts` to no longer take a function pointer as input and instead to use a boolean to determine which function needs to be called. Finally, in `onJoinPool`, scaling inputs for one type of `joinKind` was removed. We believe this is sound since the Certora Prover chooses arbitrary inputs.

Properties

(✔) invariant `solvency` : Sum of all users' BPT balance must be less than or equal to BPT's `totalSupply`.

(✔) rule `noFreeMinting` : The total supply of BPT must not increase if the total tokens held by the pool don't increase. We assume the following: * return values from `onJoinPool` and `onExitPool` are the exact amounts that are transferred to and from the pool. * rule is broken up by cases to ease the load on the prover. This rule is proven in 8 cases based on zero/nonzero `totalSupply`, on/off recovery mode, and the two types of `joinKind` : specifying exact BPT to get and specifying exact tokens to send. We assume this gives us the same coverage as proving the property in one rule.

(✔) rule `onlyOnJoinPoolCanAndMustInitialize` : `totalSupply` must be non-zero if and only if `onJoinPool` is successfully called. Additionally, the balance of the zero address must be nonzero if `onJoinPool` was successfully called.

(✔) rule `cantBurnZerosBPT` : The zero address's BPT balance can never go from nonzero to zero. We assume that no one has access to the private keys of the zero address.

(![FAILING]) rule `amplificationFactorBounded` : The amplification factor must not go past the set minimum amp and maximum amp * This rule passes on the preserve, however it fails the constructor.

(✔) rule `amplificationFactorFollowsEndTime` : After starting an amplification factor increase and calling an arbitrary function, for some e later than initial increase amplification factor must be less than or equal value set.

(✔) rule `amplificationFactorNoMoreThanDouble` : The amplification factor may not increase by more than a factor of two in a given day.

(✔) rule `amplificationFactorUpdatingOneDay` : If the amplification factor starts updating, then it must continue so for one day.

(✔) invariant `basicOperationsRevertOnPause` : All basic operations must revert while in a paused state.

(✔) rule `pauseStartOnlyPauseWindow` : If a function sets the contract into pause mode, it must only be during the `pauseWindow` .

(✔) rule `pauseStartOnlyPauseWindow` : If a function sets the contract into pause mode, it must only be during the `pauseWindow` .

(✔) rule `unpausedAfterBuffer` : After the buffer window finishes, the contract may not enter the paused state.

(✔) rule `prOtherFunctionsAlwaysRevert` : If both paused and recovery mode is active, the set functions must always revert.

(✔) rule `recoveryExitNoStableMath` : In recovery mode, exit never calls any simple math functions.

Verification of WordCodec.sol

This document describes the specification and verification of Balancer's WordCodec library using the Certora Prover. The work was undertaken from June 2022 to September 2022. The latest commit that was reviewed and run through the Certora Prover was commit 0977c144.

The scope of our verification was WordCodec.sol and its dependencies.

Assumptions and Simplifications

We were able to complete the verification of this contract without the need for any noteworthy assumptions or simplifications.

Harnessing

The contract `WordCodecHarness` inherits from the `WordCodec` library to allow the Certora Prover to interact with the library methods. We called `internal` library methods from `public` harness wrappers for the purposes of verification, as well as two helper functions which had previously been declared `private` (see Munging below). The wrappers for these helper functions were distinguished as `validateEncodingParamsUint` and `validateEncodingParamsInt`.

Munging

We munged away an import of `BalancerErrors.sol` which was colliding with an import of the same file elsewhere. We also munged both overloaded helper functions named `_validateEncodingParams` to be `internal` rather than `private` so we could call them from their respective wrappers in `WordCodecHarness`.

Properties Reversion Behavior

(✔) rule `decodeUintDoesNotRevert` : Calls to `decodeUint` must not revert.

(✔) rule `decodeIntDoesNotRevert` : Calls to `decodeInt` must not revert.

(✔) rule `insertBits192DoesNotRevert` : Calls to `insertBits192` must not revert.

(✔) rule `insertBoolDoesNotRevert` : Calls to `insertBool` must not revert.

(✔) rule `decodeBoolDoesNotRevert` : Calls to `decodeBool` must not revert.

(✔) rule `doesNotRevertImproperly` : Method calls must not revert unless the associated parameter validation reverts.

Integrity

(✔) rule `uintInsertDecodeIntegrity` : Inserting and decoding a `uint` must return the original `value`.

(✔) rule `intInsertDecodeIntegrity` : Inserting and decoding an `int` must return the original `value`.

(✔) rule `uintEncodeDecodeIntegrity` : Encoding and decoding a `uint` must return the original `value`.

(✔) rule `intEncodeDecodeIntegrity` : Encoding and decoding an `int` must return the original `value`.

(✗) rule `boolInsertDecodeIntegrity` : Inserting and decoding a `bool` must return the original `value` .

- An `offset` greater than 255 breaks `bool` insert-decode integrity by always returning the original `word` for `insertBool` and always returning `false` for `decodeBool` .
- The rule passes once `offset` is constrained to be less than 256.

Bit Independence & Constraint

(✓) rule `uintInsertBitIndependence` : If a bit changes value after inserting a `uint` , it must be within the correct range.

(✓) rule `intInsertBitIndependence` : If a bit changes value after inserting an `int` , it must be within the correct range.

(✓) rule `boolInsertBitIndependence` : If a bit changes value after inserting a `bool` , it must have the correct `offset` .

(✓) rule `insertBits192BitIndependence` : If a bit changes value after inserting with `insertBits192` , it must be within the correct range.

(✓) rule `uintEncodeBitConstraint` : If a bit is outside the correct range when encoding a `uint` , its value must be 0.

(✓) rule `intEncodeBitConstraint` : If a bit is outside the correct range when encoding an `int` , its value must be 0.

Method Equivalence

(✓) rule `uintInsertEncodeEquivalence` : Encoding a `uint` and moving the appropriate value into a given `word` must yield the same result as inserting the `uint` into that same `word` .

(✓) rule `uintEncodeInsertZeroWordEquivalence` : Inserting a `uint` into an empty `word` must yield the same result as encoding that `uint` .

(✓) rule `intInsertEncodeEquivalence` : Encoding an `int` and moving the appropriate value into a given `word` must yield the same result as inserting the `int` into that same `word` .

(✔) rule `intEncodeInsertZeroWordEquivalence` : Inserting an `int` into an empty `word` must yield the same result as encoding that `int` .

(✔) rule `uintInsertBits192InsertEquivalence` : Inserting a 192 bit `value` using `insertUint` must yield the same result as using `insertBits192` .

Decoding from Zero

(✔) rule `uintDecodeFromZero` : Decoding a `uint` from a zero `word` must yield 0.

(✘) rule `intDecodeFromZero` : Decoding an `int` from a zero `word` must yield 0.

- A `bitLength` of 0 yields a value of -1 instead of 0.
- The rule passes once `bitLength` is constrained to be greater than zero.

(✔) rule `boolDecodeFromZero` : Decoding a `bool` from a zero `word` must yield false.