CERTORA

Formal Verification of Balancer V2

Summary

This document describes the specification and verification of **Balancer** using Certora Prover. The work was undertaken from **February 15th to April 22nd**. The latest commit that was reviewed and run through the Certora Prover was **8b5773510dfc7a94d4eef3e22d1de50becb0250d**.

The scope of our verification was the **Vault** contract.

The Certora Prover proved the implementation of the **Vault** is correct with respect to the formal rules written by the **Balancer** and the Certora teams. During the verification process, the Certora Prover discovered various issues in the code, some major. All issues were corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is constantly working to remedy these limitations. In this document, we describe the most interesting issues that we found in the Vault contract, and the formal descriptions of the specifications we used to find these issues.

Discovered Issues

Issue	Providing the same token address multiple times in the flash loan's input token address array causes tax evasion and insolvency.
Severity	**High**
Rule(s) broken	Flash loan additivity
Description	Instead of calling a flash loan on token T for amount K with [T], [K] as input parameters, calling the flash loan with [T, T, ...] N times and amounts [N/K, N/K, ...] N times, the user only pays 1/N of the full flash loan fee. The system still thinks it collected the entire fee, making it insolvent. Consequently, the vault owner can withdraw money that was incorrectly counted as collected fees at the expense of users.

	The effects of this attack can be exponentially increased. $\forall$ i, amounts[i+1] = $\frac{1}{1+1/f}$ * amounts[i]. The total fees we pay this way is $\frac{1}{(1+1/f)^n - 1}$, where *f* is the flash loan fee ratio (a number between 0 and 0.01) and *n* is the length of the tokens array. Using this scheme, we decrease the fees exponentially by *n*. For example, with an array of length ten and a maximal fee of 1%, we only need to pay a share of $9*10^{-21}$ of the amount loaned.
Fix	The token input array must now be sorted in strictly ascending order. This prevents using the same token address more than once at the input array.

Issue	Funds invested in pools are locked when the system is paused
Severity	**High**
Rule(s) broken	The integrity of the emergency recovery scheme
Description	When the system is paused, users should take their funds out of the system: withdraw everything from their internal balance and exit all pools they ever joined. Calling *exitPool()* always reverted during a pause. This happened because of a call to *_joinOrExit()*, a shared function for joins and exits with a *TemporarilyPausable* modifier, that prevents transactions temporarily. There was no way to retrieve user funds stored inside pools during a pause, severely undermining the emergency recovery scheme.
Fix	Move the *TemporarilyPausable* modifier from the shared *_joinOrExit()* function to the external function *joinPool()*.

Issue	No withdrawal tax was charged when users withdrew funds from their internal balance if they swapped with a pool that block.
Severity	**Medium**
Rule(s) broken	Withdrawal tax evasion
Description	There was a mechanism in place to exempt the withdrawal fee if a user used their internal balance temporarily. The intended tax-free scenario was when a user deposited money to their internal balance, used it in a swap, and withdrew it, all during the same block. However, the mechanism also exempted withdrawals if a swap happened during the same block, regardless of when the funds were deposited. Before this mechanism was introduced, a user could still avoid paying the withdrawal fee by constructing a custom pool with a backdoor, skipping the withdrawal step entirely. The solution to that problem was to set the withdrawal fee so low that it would not be worth the hassle. With this mechanism, even honest users will evade paying the withdrawal fee, and the effort required for intentional tax evasion is minimal.
Fix	Withdrawal fees are no longer charged.

Issue	Wrong error reported when an unregistered pool's id is given to a swap function.
Severity	**Low**
Problem	Wrong error message
Description	When a pool id for an unregistered pool was passed to a swap, we reverted with the "TOKEN NOT REGISTERED" error. All pool specializations share this behavior.

Fix	A check if the pool is registered was added. If it is not, we revert with the error “POOL_NOT_REGISTERED”.

Issue	Obscure revert with no reason string for most illegal pool ids in swaps.
Severity	**Low**
Problem	Wrong error message
Description	When swapping with a pool, the pool’s specialization was checked before checking whether that pool was registered. The pool’s id contains two specialization bytes, but the specialization enum has only three values. If the calculated specialization fell out of the enum range, we would revert without a reason string.
Fix	An explicit check of whether the calculated pool specialization is in the legal range was added. If it is not, we revert with the error “INVALID_POOL_ID”.

Issue	No bulk role revoke functions.
Severity	**Low**
Problem	Compromised addresses recovery scheme.
Description	An address can be granted many roles, say by the *grantRoles()* or *grantRolesToMany()* functions. If that address is later compromised, we want to have an easy and efficient way to revoke all of its roles. Reverse operations to *grantRoles()* and *grantRolesToMany()* did not exist; roles could be revoked one at a time, individually.
Fix	Bulk revoke functions *revokeRoles()* and *revokeRolesFromMany()* were added.

Issue	*TokenRegistered* event emitted does not reflect possible reordering of tokens of a TwoTokensPool.
Severity	**Low**
Rule(s) broken	The order of tokens in a pool is constant for all operations besides *deregisterTokens()*.
Description	A user may register any two different non-zero token addresses to a *TwoTokensPool*. The emitted event will report the token addresses in the order the user inserted them. When a user gives two descending token addresses as input, the vault will sort them in ascending order. The vault requires the token addresses to be sorted in ascending order for any subsequent action. However, the emitted event does not indicate the order change to the user.
Fix	Require the token addresses to be input in (strictly) ascending order.

Issue	No reverts when there are more asset managers than tokens in registerTokens().
Severity	**Low**
property broken	Each registered token must have an asset manager.
Description	At PoolRegistry.sol in *registerTokens()*, we did not verify that the *tokens* and *assetManagers* input arrays had equal lengths. If the array of tokens was longer, we reverted to an out-of-bounds exception. If there were more managers than tokens, the last reminding managers in the input array were ignored.
Fix	Add `InputHelpers.ensureInputLengthMatch(tokens.length, assetManagers.length)` to *registerTokens()*.

Issue	Wrong error message when the first swap in a batch swap consisting of two or more swaps had an amount of zero.
Severity	**Low**
Problem	Wrong error message, possible optimization
Description	If the first swap in a batch swap consisting of two or more swaps had an amount of zero, it reverted with an error message “MALCONSTRUCTED_MULTIHOP_SWAP”.
Fix	The if condition changed from (*swaps.length* > *1*) to (*i* > *0*). Now, if the first token has an amount of zero, we revert with the error “UNKNOWN_AMOUNT_IN_FIRST_SWAP”. We also fail faster in this case, reducing gas costs.

Issue	Wrong error message when the first token given to a flash loan is the zero address token.
Severity	**Low**
Problem	Wrong error message.
Description	When the first token in the input token array of a flash loan was the zero token, we reverted with the error “UNORDERED_TOKENS”. That happened even if the tokens were ordered, and even if there was only a single token in the input array.
Fix	An explicit check of whether a token is the zero token or not was added. A helper function was added to check if the array is sorted. The helper function ignores arrays of length less than two.

Issue	Inconsistent error messages between *WeightedPool* and *StablePool*.
Severity	**Low**
Problem	Inconsistent error messages
Description	Different error messages were reported by different pool types when

	we insert two input arrays of unequal length in the functions *_onInitializePool* and *_joinExactTokensInForBPTOut*. *StablePool* used a helper library function, while *WeightedPool* implemented the check and returned a different error message.
Fix	The helper function was used in *WeightedPool* as well for consistency.

Issue	Array out of bounds error when joining or exiting a *TwoTokensPool* with no registered tokens.
Severity	**Low**
Problem	Inconsistent behavior between pool types; bad error message
Description	When we joined or exited a *TwoTokensPool*, the code referenced indices 0 and 1 in the *tokens* input array. However, a *TwoTokensPool* might not have any registered tokens. When a user tried to join or exit a *TwoTokensPool* with no registered tokens and provided an empty array as an input, we did not fail the token correctness check for the pool. However, we referenced illegal indices and reverted with an array out of bounds error. This is not just a confusing error message for the user; it was also a different behavior than pools of type *MinimalSwapInfoPool* and *GeneralPool*, which did not fail in this scenario. Note that joining or exiting pools with no registered tokens should not affect the system.
Fix	As a part of the token validation function, we also ensure that the pool has at least one registered token and report an appropriate error message if it does not.

Issue	Trivial *require* statements.
Severity	**Recommendation**
Problem	Wasted resources - gas, bytecode size
Description	Two *require* statements in BalancerPoolToken.sol could never fail because of the if condition they were inside. These statements had no functional effect on the code but increased the contract's byte code, run time, and gas costs.
Fix	The trivial *require* statements were removed.

Issue	Unused variable
Severity	**Recommendation**
Problem	Hard to read code
Description	In InternalBalance.sol, we declared a variable *amountToSend* as an alias to the variable amount. However, the variable amount was never used.
Fix	The variable was renamed as the alias name.

Issue	Unused interface
Severity	**Recommendation**
Problem	Wasted resources - bytecode
Description	No contract implemented the interface *IAuthorizer*
Fix	The *Authorizer* contract now implements the *IAuthorizer* interface.

Issue	Unused functions in helper libraries
Severity	**Recommendation**
Problem	Wasted resources - bytecode, code readability
Description	There are several libraries in the project which are based on OpenZeppelin libraries. They were altered to include unique error codes and other optimizations. However, most files included functions and structs that were never used by any project file. The relevant files were: AccessControl.sol, Address.sol, EnumerableSet.sol, Context.sol, Counters.sol, and SafeCast.sol
Fix	All unused functions and structs were removed.

Issue	Unused file
Severity	**Recommendation**
Problem	Wasted resources - bytecode, code readability
Description	The file contracts/lib/openzeppelin/EnumerableMap.sol was never used. contracts/lib/helpers/EnumerableMap.sol was used instead.
Fix	The unused file contracts/lib/openzeppelin/EnumerableMap.sol was deleted.

Issue	Unused inheritances
Severity	**Recommendation**
Problem	Error prevention
Description	The *Fees* contract inherited from *ReentrancyGuard* and *VaultAuthorization* despite not using any of their functions or variables.

	The *InternalBalance* contract inherited from *Fees* but could inherit from *VaultAuthorization* only.
Fix	Unused inheritances were removed.

Issue	Reentrancy to modifiers
Severity	**Recommendation**
Problem	Error prevention
Description	There are several function modifiers in the code. One of them is *nonReentrant*, which prevents calling the same code twice (say, from within a flash loan). However, if we do not list *nonReentrant* as the first modifier, reentrancy is possible to the modifiers' code that appears before it. All modifiers were reentrancy safe, but this could potentially prevent future issues at no cost.
Fix	The *nonReentrant* modifier is now always the first modifier for all functions.

Issue	Duplicated statements with no effect
Severity	**Recommendation**
Problem	Wasted resources - bytecode size, run time, gas; readability
Description	In the file PoolAssets.sol, there were two duplicate if-else statements. They had no functional change but increased the bytecode size of the contract, as well as the run time and hence gas costs.
Fix	The duplicate statements were removed.

Issue	Documentation fixes
Severity	**Recommendation**
Problem	Readability
Description	Several documentation pieces were incorrect and did not describe the actual behavior of the code. We highlighted dozens of typos and grammatical mistakes across all code files. In many cases, the documentation of similar parts was inconsistent across different files or functions. We corrected inconsistent variable and function naming.
Fix	The documentation was updated accordingly.

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 any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express 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

1. ✔️ indicates the rule is formally verified on the latest commit.
2. ✔️* indicates we verified the rule on a simplified version of the code. The simplified version has abstractions to simplify fee calculation, pool specialization check, signature approval, and pool balance management (no managed funds, everything is cash). The rules were not proven for batchSwap but were proven on swap. We did not prove these rules on queryBatchSwap, but that function behaves as a view function and is of little interest.
3. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
   The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim$op C_2, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of Vault

Balancer's *vault* is a system that facilitates currency *swaps* between users and arbitrary liquidity *pools*. Anyone can write a pool contract and add it to the system. Users can join a pool and give it funds as *liquidity providers*. The pool logic is arbitrary. In particular, a pool can be *malicious* and steal the funds of the liquidity providers. Balancer would provide a white list of safe pools, and only they would be available in the user interface. Under no circumstances should a pool affect other pools or users in the system.

The main contract of the system is the *vault*. It is the only interface through which the users and the pools interact. The vault has many components, divided into sub-contracts. The scope of our verification was the vault and all of its components.

Users can deposit money into their *internal balance*. This balance acts like a specialized wallet that is gas optimized for swaps with the pools. Users can deposit and withdraw funds to their internal balance or transfer money to another user's internal balance freely without fees.

A user can *register* any pool that implements the pool interface. The pool's owner can *register* and *deregister* tokens to that pool. Tokens can only be deregistered if the pool contains no funds. Users can *join* pools as liquidity providers by providing some of that pool's registered tokens. To take deposited funds away from the pool, a user can *exit* the pool.
For gas optimization, there are three types of pools. *Two Token Pools* can only contain exactly two tokens. *Minimal Swap Info* pools only care about the balances of the two tokens being swapped to calculate the swap rate. *General Pools* cover the rest.
A pool may have an *asset manager*. Asset Managers can take money from the pool's *cash*, the funds deposited into the pool by users joining it, and turn it into *managed* funds. The pool's manager has total control over the pool's managed funds, and users cannot withdraw them.

The pool supports *flash loans*. A user can withdraw all the money the vault possesses, provided they can return all of it at the same transaction, plus a flash loan fee.

The central feature of the vault is its efficient implementation of swap operations. A user can *batch swap* with several pools in sequential order at the same transaction cheaply. The exact amount of funds a user will get back from the pools is unknown to the vault in advance. Therefore, there are two types of swaps. In a *given in swap*, the user specifies the maximal amount of tokens they are willing to pay. In a *given out swap*, the user selects the minimal amount of funds they are ready to accept in the exchange.
The vault supports trade with all tokens, as well as ETH. The vault and the pools never hold ETH; it is wrapped as WETH in a transparent way to the user.

ProtocolFeesCollector is an external contract that calculates the fees required by vault operations and stores the collected fees. Users pay a flash loan fee after a successful flash loan. The users also pay *swap fees* at every swap. The pools calculate the amount of money to collect as swap fees, and the vault trusts the pools to report accurate rates.

Users can allow other users to act on their behalf. A user can approve another address as their *relayer*, or if they give that user a valid signature. Additionally, the vault has an external *Authorizer*. A user may only act on another user's behalf if they are both approved by the user and the external authorizer authorizes them.

As a defense mechanism against bugs, the vault can be *paused*. While a vault is paused, users may withdraw their internal balance of funds given as liquidity, and no other action is allowed. The vault can be paused only during the first three months after deployment. The pause ends four months after deployment, so it is, at minimum, one month long.

Functions

The functions are presented in alphabetical order.

batchSwap(kind, swaps[], assets[], limits[]): int256[]

Performs a series of swaps, in sequential order. For more details, see the swap function.

deregisterTokens(pool, tokens[])

Deregisters each token in *tokens* from *pool*.

doubleExitPool(pool, recipient, tokenA, tokenB, amountA, amountB)

The caller takes funds out of a pool they provided liquidity for. The user withdraws *amountA* tokens of type *tokenA*, and *amountB* tokens of type *tokenB*. The funds are transferred to *recipient*.

doubleFlashLoan(tokenA, tokenB, amountA, amountB)

The caller loans *amountA* tokens of type *tokenA*, and *amountB* tokens of type *tokenB*. They must return them in the same transaction, plus fees, or the loan will revert.

doubleJoinPool(pool,sender,tokenA, tokenB, amountA, amountB)

The user joins the pool as a liquidity provider by providing *amountA* tokens of type *tokenA*, and *amountB* tokens of type *tokenB*. The money will be taken from *sender*, provided that the user is authorized to do so.

getACollectedFee(token) : uint

Returns the amount of tokens of type *token* the *ProtocolFeesCollector* has collected.

getAnInternalBalance(user, token) : uint

Returns the amount of token of type *token* the *user* has in their internal balance.

getFlashLoanFeePercentage() : uint

Returns the fee percentage charged for flash loans.

getGeneralPoolBalance(pool, token) : uint

Returns the amount of tokens of type *token* contained in *pool*. *pool* must be a pool with specialization *General*.

getMinimalSwapInfoPoolBalance(pool, token) : uint
Returns the amount of tokens of type *token* contained in *pool*. *pool* must be a pool with specialization *MinimalSwapInfo*.

getPoolAddress(poolId) : address
Returns the address of the pool with identifier *poolId*.

getReceiveAssetCounter() : uint
Auxiliary spec function; returns how often *receiveAsset()* has been called.

getSwapFeePercentage() : uint
Returns the fee percentage charged for swaps.

getTokenBalance(user, token) : uint
Gets the amount of tokens of type *token* that *user* has in their external balance.

has_valid_signature(user) : bool
Auxiliary spec function; returns *true* if a *user* has a valid signature to perform an operation, and *false* otherwise.

hasApprovedRelayer(user, relayer) : bool
Returns *true* if *relayer* is approved to be a relayer for contract instance *user*, and false otherwise.

isAuthenticatedForUser(user) : bool
Returns *true* if *user* has approved *msg.sender* as a relayer, and the external authorizer also authenticated *msg.sender*; returns *false* otherwise.

isPoolRegistered(poolId) : bool
Returns true if a pool with identifier *poolId* has been registered.

isTokenRegisteredGeneral(pool, token) : bool
Returns true if *token* is one of the registered tokens in the General Pool *pool*, false otherwise.

isTokenRegisteredMinimalSwap(pool, token) : bool
Returns true if *token* is one of the registered tokens in the Minimal Swap Info Pool *pool*, false otherwise.

manageUserBalance(ops[])

Performs a set of user balance operations, which can involve internal balance (deposit, withdraw or transfer) and plain ERC20 transfers using the Vault's allowance.

receiveAsset(asset, amount, sender, fromInternalBalance)

The vault receives *amount* tokens of type *asset* from *sender*. The asset can be either a token or native ether. If *frominternalBalance* is true, money will be taken from *sender*'s internal balance. If the balance is lower than *amount*, the difference will be taken from the external balance of the sender. If *frominternalBalance* is false, all of *amount* will be taken from *sender*'s external balance.

registerTokens(pool, tokens[], assetManagers[])

Registers each token type in *tokens* to *pool*. Each token has a matching asset manager in the same index in *assetManagers*.

setRelayerApproval(user, relayer, allowed)

If *allowed* is true, allows *relayer* to be a relayer for *user*. If *allowed* is false, disallows *relayer* to be a relayer for *user*.

singleExitPool(pool, recipient, token, minAmountOut)

The *recipient* takes funds out of a pool they provided liquidity for. The user withdraws *amount* tokens of type *token*, at least *minAmountOut*.

singleJoinPool(pool, sender, token, maxAmountIn)

The user joins the pool as a liquidity provider by providing *maxAmountIn* or less tokens of type *token*. The money will be taken from *sender*, provided that the user is authorized to do so.

swap(singleSwap, funds) : uint

Performs a swap between two pools. *singleSwap* is a (nested) struct containing, among others, the fields *sender, recipient*, and *amount*. A swap operation sends some amount of tokens of some token type from the *sender* pool to the *recipient* pool and sends another amount of tokens of another type from the *recipient* pool to the *sender* pool. The exact amounts of tokens sent are computed from the *amount* field according to some modifiers like exchange rate.

toPoolId(poolAdr, specialization, nonce) : bytes32
Gets an address of a pool contract *poolAdr*, that pool's *specialization*, and a *nonce*, and returns a matching identifier for the pool.

withdrawCollectedFees()
Withdraws all collected fees from *ProtocolFeesCollector*.

Global Constants

MAX_PROTOCOL_SWAP_FEE_PERCENTAGE
Maximum fee percentage charged for a swap

MAX_PROTOCOL_FLASH_LOAN_FEE_PERCENTAGE
Maximum fee percentage charged for a flash loan

feesCollector
Instance of the vault's ProtocolFeesCollector contract

vault
Instance of the vault contract

Properties

All properties were verified for a bounded loop size of two unless noted otherwise.

Fees

1. All fees are bounded from above by 100%.
   1.1 The swap fee is smaller than the maximal protocol swap fee. ✔

   (invariant-rule: feeIntegrity.spec:swap_fee_integrity)

   `getSwapFeePercentage() ≤ MAX_PROTOCOL_SWAP_FEE_PERCENTAGE`

   1.2 The flash loan fee is smaller than the maximal flash loan fee. ✔

(invariant-rule: feeIntegrity.spec:flash_loan_fee_integrity)

```
getFlashLoanFeePercentage() ≤ MAX_PROTOCOL_FLASH_LOAN_FEE_PERCENTAGE
```

1.3 The maximal fees are smaller than 100%. ✔

(invariant-rule: feeIntegrity.spec:fee_integrity)

```
MAX_PROTOCOL_SWAP_FEE_PERCENTAGE < 1 ∧
MAX_PROTOCOL_FLASH_LOAN_FEE_PERCENTAGE < 1
```

2. A user cannot decrease the collected fees of the vault. The only exception is the vault's owner, who can withdraw the collected fees. ✔*

(rule: general_spec.spec:increasingFees)

```
{ feePre = getACollectedFee(tokens) ∧
  op ≠ withdrawCollectedFees }
   op()
{  feePre = getACollectedFee(tokens) }
```

Pool registration

3. Reconstruction of pool address from pool id. ✔

(rule: poolregistry.spec:toPoolIdAndThenGetPoolAddressIsNeutral)

```
getPoolAddress(toPoolId(poolAdr, specialization, nonce)) = poolAdr
```

4. A token can be registered to a Pool only if that pool is already registered:

4.1 General Pool ✔*

(invariant rule: general_spec.spec:tokensGeneralPoolRegistration)

```
isTokenRegisteredGeneral(poolId, token) ⇒ isPoolRegistered(poolId)
```

4.2 Minimal Swap Info Pool ✔*

(invariant rule: general_spec.spec:tokensMinimalSwapInfoPoolRegistration)

```
isTokenRegisteredMinimalSwap(poolId, token)
      ⇒ isPoolRegistered(poolId)
```

5. The pool's balance is greater than zero only if the pool is registered:

   General Pools ✔*

   (rule: general_spec.spec:general_pool_positive_total_if_registered)

   ```
   getGeneralPoolBalance(poolId, token) ≠ 0
         ⇒ isPoolRegistered(poolId)
   ```

   Minimal Swap Info Pools ✔*
   (rule:general_spec.spec:minimal_swap_info_pool_positive_total_if_registered)

   ```
   getMinimalSwapInfoPoolBalance(poolId, token) ≠ 0
         ⇒ isPoolRegistered(poolId)
   ```

Internal balance

6. User A can decrease user B's internal balance only if they are authenticated to do so by both the external system authorizer and the user themselves. ✔*

   (rule: internalBalance.spec:only_authorizer_can_decrease_internal_balance)

   ```
   { balancePre = getAnInternalBalance(user, token) }
         op()
   { getAnInternalBalance(user, token) < balancePre
      ⇒ (isAuthenticatedForUser(user) ∨ has_valid_signature(user)) }
   ```

7. Effects of external and public functions on the internal balance of users. ✔*

   Most methods cannot affect the internal balance of a user in any way. Exceptions include the joinPool (exitPool) operations, which can only decrease (increase) the caller's balance. All exceptions are described in the table below.

Method	Increase	Decrease
`manageUserBalance`	Possible	Possible
`swap`	Possible	Possible
`batchSwap`	Possible	Possible
`joinPool`	Not Possible	Possible
`exitPool`	Possible	Not Possible
`<all other operations>`	Not Possible	Not Possible

Note that *joinPool* here stands for both the *singleJoinPool* and *doubleJoinPool* methods, and analogously for *exitPool*.

(rule: internalBalance.spec:internalBalanceChanges)

```
{ balancePre = getAnInternalBalance(user, token) }
      op()
{ balancePost = getAnInternalBalance(user, token) ∧
  (op = joinPool ⇒ balancePost <= balancePre) ∧
  (op = exitPool ⇒ balancePost >= balancePre) ∧
  (op ∉ {joinPool, exitPool, swap, batchSwap, manageuserBalance}
   ⇒ balancePost = balancePre) }
```

Authentication

8. Consistent updating and querying of relayer permission ✔*

   (rule: general_spec.spec:setRelayerApprovalIntegrity)

   ```
   { }
         setRelayerApproval(user, relayer, allowed)
   { hasApprovedRelayer(user, relayer) = allowed }
   ```

9. The vault never gives relayer approvals: ✔*

(invariant-rule: internalBalance.spec:vault_has_no_relayers)

```
hasApprovedRelayer(vault, user) = false
```

Join and Exit Pool

10. JoinPool cannot lose the vault money (the external ERC balance of the vault cannot decrease after a joinPool). ✔*

    (rule: joinExitPool.spec:[harmlessJoinPoolGeneralMinimal|harmlessJoinPoolTwoTokens])

    ```
    { balancePre = getTokenBalance(vault, token) }
          doubleJoinPool(pool, sender, tokenA, tokenB, amountA, amountB)
    { balancePre ≤ getTokenBalance(vault, token) }
    ```

11. A user cannot profit, in any token, from JoinPool ✔*

    (rule: joinExitPool.spec:noJoinPoolUserProfit)

    ```
    { balancePre = getTokenBalance(user, token) ∧
      user ∉ {vault, feesCollector} }
          doubleJoinPool(pool, sender, tokenA, tokenB, amountA, amountB)
    { balancePre ⇒ getTokenBalance(user, token) }
    ```

12. A user's total profit, of both ERC and internal balance, from an ExitPool of a single token from a General pool, must be at least minAmountOut ✔*

    (rule: joinExitPool.spec:exitPoolMinUserProfit)

    ```
    { ercPre =  getTokenBalance(sender, token) ∧
      internalBalancePre = getAnInternalBalance(sender, token) }
          singleExitPool(pool, sender, recipient, token, minAmountOut)
    { (  getTokenBalance(sender, token) - ercPre ) +
      ( getAnInternalBalance(sender, token) -  internalPre ) ≤ minAmountOut
    }
    ```

13. exitPool cannot increase the pool's balance:
    General Pool ✔*
    Minimal Swap Info Pool ✔*

(rule: joinExitPool.spec:[exitPoolProfitabilityGeneralMinimalPool])

```
{ feePre = getACollectedFee(token) }

      doubleExitPool(pool, recipient, tokenA, tokenB, minAmountOutA,
minAmountOutB)

{ feePre ≤ getACollectedFee(token) }
```

Miscellaneous

14. *receiveAsset()* can be called at most once per asset in each transaction. In other words, we look at e.msg.value only once per batch swap. ✔*

    (rule: eth.spec:receive_asset_called_at_most_once_per_token)

    ```
    { counterPre = getReceiveAssetCounter() }

          op()

    { getReceiveAssetCounter() <= counterPre + 1 }
    ```

15. Profitability of flash loans - the vault cannot lose any amount of tokens of any type due to a successful flash loan. ✔*

    (rule: flashLoanAdditivity.spec:flashLoanProfitability)

    ```
    { balancePre = getTokenBalance(user, token) }

          doubleFlashLoan(tokenA, tokenB, amoutA, amountB)

    { balancePre <= getTokenBalance(user, token) }
    ```