Security Audit Report

Quipuswap Stableswap AMM

Delivered: June 22nd, 2022

Prepared for the Madfish Solutions Inc. by

runtime verification

Contents

Summary

Runtime Verification Inc. conducted a security audit funded by Tezos Foundation from April 18, 2022 to June 6, 2022 on the Quipuswap Stableswap automated market maker (AMM) contract developed by Madfish Solutions, Inc. During the audit, one medium severity issue and 12 low severity/informative issues were found relating to asset security, input validation, and best practices. In response, the Madfish team produced a new version of the Quipuswap Stableswap contract that addressed the 1 medium severity finding and incorporated nearly all of our general recommendations. See a detailed summary in the table below:

Issue	Severity	Difficulty	Status
Divest Entrypoints May Burn All Shares with Tokens Left in Pool	Medium	Low	Fixed
Certain Entrypoints Can Take Tokens without Providing Value	Low	Low	Fixed
Imprecise Over-approximations of Final Balances	Low	Low	Fixed
Three Contract Computations Unnecessarily Drop Precision	Low	Low	Fixed
Liquidity Provider Fee Calculations Can Diverge from Specification	Low	Low	Fixed
Pool Creation Cost Calculation Can Diverge from Specification	Low	Low	Fixed
Null-valued Storage Map Entries are Not Cleared	Low	High	Acknowledged
Duplicate Definition of `get_tokens_from_param`	Informative	N/A	Fixed
FA2 Transfers From Undefined LP Tokens are Not Rejected	Informative	N/A	Fixed
Update Operators Entrypoint Permits Owners as Operators	Informative	N/A	Fixed
Minor Documentation Issues	Informative	N/A	Fixed
Minor Constant Definitions Issues	Informative	N/A	Fixed
Code Style Recommendations	Informative	N/A	Fixed

Scope

The target of the audit is a subset of the smart contract source files located in the Quipuswsap Stableswap Github repository at commit id 13e92f2. The subset was defined by those files which were included in source builds of the system that defined the `FACTORY` macro. In particular, the source files reviewed were (all under the `contracts/` directory):

- `main/dex.ligo`
- `main/dex4factory.ligo`
- `main/factory.ligo`
- `partials/admin/lambdas.ligo`
- `partials/admin/methods.ligo`
- `partials/admin/types.ligo`
- `partials/common_types.ligo`
- `partials/constants.ligo`
- `partials/constants_test.ligo`
- `partials/dev/lambdas.ligo`
- `partials/dev/methods.ligo`
- `partials/dev/types.ligo`
- `partials/dex_core/factory/helpers.ligo`
- `partials/dex_core/helpers.ligo`
- `partials/dex_core/lambdas.ligo`
- `partials/dex_core/math.ligo`
- `partials/dex_core/methods.ligo`
- `partials/factory/views.ligo`
- `partials/utils.ligo`
- `partials/dex_core/storage.ligo`
- `partials/dex_core/types.ligo`
- `partials/dex_core/views.ligo`
- `partials/errors.ligo`
- `partials/fa12/types.ligo`
- `partials/fa2/helpers.ligo`
- `partials/fa2/lambdas.ligo`
- `partials/fa2/methods.ligo`
- `partials/fa2/types.ligo`
- `partials/fa2/views.ligo`
- `partials/factory/deploy.ligo`
- `partials/factory/helpers.ligo`
- `partials/factory/lambdas.ligo`
- `partials/factory/methods.ligo`
- `partials/factory/storage.ligo`

The above criteria *excluded* two contract source files:

- `partials/admin/standalone/lambdas.ligo`
- `partials/dex_core/standalone/helpers.ligo`

The audit is limited in scope to the LIGO smart contract source code only. Off-chain and client side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement and are assumed to be correct. If we happen to observe any issues outside of the stated scope, we will report them, but no guarantees are made about out of scope entities.

In response to the audit, Madfish Solutions produced a new version with commit id 4a99a80.

Methodology

Although the manual code review cannot guarantee that all potential security vulnerabilities are found (as mentioned in the Disclaimer), we have adhered to the following methodology to make our audit as rigorous as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Then we carefully checked if the code is vulnerable to known security issues and attack vectors. We took particular care to analyze the stableswap curve curve, which the contract uses as its pricing function. Though this curve has become quite popular, we could not find any rigorous mathematical analyses of it. During our audit, we thoroughly analyzed the stableswap curve to ensure that it satisfies the basic requirements of AMM pricing functions. We then checked that the contract correctly implements this pricing curve. During this process, we searched for as many potential sources of error as possible, including rounding error, convergence failure, edge case behavior, etc.

Security Requirements

When analyzing automated market makers, there are several key properties that must hold for the security of the contract. Some of these are properties of blockchain itself while others are properties of the AMM contract or the contracts that it interacts with. We list the most important requirements here:

1. A contract's storage must only be writable via invoking that contract.
2. All FA1.2/FA2 contracts that the AMM interacts with properly conform to the appropriate standard, e.g., the `transfer` function must properly send funds.
3. Each of the AMM's FA1.2/FA2 token balances must not *decrease* unless the AMM calls the token contract's `transfer` function.
4. The AMM contract correctly implements its pricing function and correctly invokes the appropriate FA1.2/FA2 contracts.

We also make two other remarks. Firstly, the mathematics of the contract were partially validated using a Python-based simulator of the Tezos Michelson interpreter developed by Baking Bad. Our trust base thus extends to Tezos Michelson interpreter simulator code. In particular, we require that its calculations agree with the actual OCaml-based implementation of the Tezos Michelson interpreter. Secondly, logic errors related to reentrancy are a common source of security issues in smart contracts. Like the original stableswap-based AMM contract implementation, the Quipuswap Stableswap AMM contract does not have reentrancy locks. Since the contract permits reentrancy, for contract security to hold, it must be reentrant safe. During the audit, we could not find any reentrancy exploits.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and the preparers of this report note that it excludes a number of components critical to the operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

Divest Entrypoints May Burn All Shares with Tokens Left in Pool

[Severity: Medium | Difficulty: Low | Category: Asset Security]

Suppose that a pool is created that, at some point in time, has a single liquidity provider. In that case, it is possible that:

1. the liquidity provider calls the entrypoint `User_action.Use_dex.Divest_one_coin` electing to burn all of their shares:
2. the contract correctly burns all shares for the sole liquidity provider and sends them the appropriate funds;
3. . . . *but* the pool has assets which remain in it which now belong to no one.

If this happens, anyone can buy these remaining assets at below market price by immediately doing the following:

1. calling the `User_action.Use_dex.Invest` entrypoint and minting at least one liquidity share (without considering fees, since fees are not levied in case there are currently zero minted liquidity shares);
2. calling the `User_action.Use_dex.Divest` entrypoint and burning the shares that were just minted;
3. the contract will then burn all remaining shares and send all remaining dex tokens to the user.

The problem here is that, the total amount of minted shares should only be 0 whenever there are no assets left in the pool. Otherwise, a sole liquidity provider may unwittingly donate any remaining pool assets, which they *could have claimed*, to any future liquidity provider.

Recommendation: After divestment, check that the total supply of liquidity shares is 0 if and only the asset reserves are fully drained. Revert the operation if this condition does not hold.

Status: This issue was fixed in the latest version of the contact.

Certain Entrypoints Can Take Tokens without Providing Value

[Severity: Low | Difficulty: Low | Category: Asset Security]

Due to fees or rounding errors, entrypoints:

- `User_action.Use_dex.Invest`
- `User_action.Use_dex.Divest_one_coin`

can accept tokens/shares from a user *without* minting any LP shares/sending any tokens back to the user.

This occurs because a parameter (i.e. `min_mint_amount` or `min_amount_out`), which controls the minimum amount of liquidity shares minted or tokens purchased via the entrypoint, is able to be set to zero.

Recommendation: To protect users from unwittingly donating tokens to the contract, update these entrypoints to reject invocations which specify a minimum emitted value of 0.

Status: This issue was fixed in the latest version of the contact.

Imprecise Over-approximations of Final Balances

[Severity: Low | Difficulty: Low | Category: Asset Security]

The entrypoints:

- `User_action.Use_dex.Swap`
- `User_action.Use_dex.Divest_one_coin`

internally call `get_y` and `get_y_D` respectively which both call `calc_y`. In each case, the computed value of `y` represents either:

- the new balance of the swap output token less the swap output:
- the new balance of the divestment output token less the divestment.

while `dy` represents the difference the original and final token balances. It is important when rounding to over-approximate the value of `y` and under-approximate `dy` (i.e., leave *more* money in the dex and give *less* money to swapper/liquidity provider) to ensure that dex users cannot exploit rounding error to make a profit.

The stableswap algorithm which is implemented by the contract computes `y` using Newton's method. After applying Newton's method, the contract imprecisely over-approximates `y` by adding 1. Ideally, we would like to *provably over-approximate* the final value of `y`.

To do this, we can control rounding in the Newton method used to calculate `y`. The Newton method loop body looks like follows:

$$y' = \frac{y^2 + c}{2y + b - D}$$

where the loop continues to iterate until it converges, i.e., until $|y' - y| < \delta$ for some $\delta > 0$ and we have the abbreviations:

- $c = \frac{D^{n+1} * A_{precision}}{A * n * n^n * \prod X / \{x_j\}}$;
- $b = \sum X / \{x_j\} + \frac{D * A_{precision}}{A * n}$.

and where all constants are non-negative. In order to over-approximate the final value of `y`, we want division:

- performed in the numerator (the computation of `c`) to round up (ceil division)
- performed in the denominator (the computation of `b`) to round down (floor division)
- where the denominator is taken from the numerator to round up (ceil divsion)

See the section **Newton Method Analysis** in the appendix for a full derivation of the Newton method loop body as well as a justification for the why the Newton method will always over-approximate `y`.

With this *provable* over-approximation of `y`, we no longer need to rely on the imprecise over-approximation that just adds 1 (or equivalently, subtracts 1 from `dy`). That is, we can remove the following imprecise approximations:

1. the subtract by 1 performed by `perform_swap`:

   ```
   const dy = nat_or_error(xp_j - y - 1, Errors.Math.nat_error); // -1 just in
       case there were some rounding errors
   ```

2. the subtract by 1 by performed by `calc_withdraw_one_coin`:

   ```
   dy := nat_or_error(dy - 1, Errors.Math.nat_error) / precisions_i; //#
       Withdraw less to account for rounding errors
   ```

Recommendation: Replace existing imprecise over-approximation of the Newton method with the provably over-approximating variant of the Newton method shown above to compute the final value of `y`. Verify via empirical testing that the updated Newton method method converges quickly enough.

Status: This recommendation was adopted in the latest version of the contact.

Three Contract Computations Unnecessarily Drop Precision

[Severity: Low | Difficulty: Low | Category: Asset Security]

In `partials/dex_core/math.ligo`, the three functions:

- `get_D`
- `get_y`
- `get_y_D`

unnecessarily drop the precision of their computed results. This occurs in three loops:

- the `count_D_P` loop

  ```
  function count_D_P(
    const accum  : nat;
    const i      : nat * nat)
                 : nat is
    accum * d_const / (i.1 * tokens_count);
  const d_P = Map.fold(count_D_P, xp, tmp.d);
  ```

- the first instance of the `prepare_params` loop

  ```
  function prepare_params(
    var accum       : record [ s_: nat; c: nat; ];
    const entry     : nat * nat)
                    : record [ s_: nat; c: nat; ] is
    block {
      var   _x  : nat := 0n;
      const iter: nat = entry.0;
      if iter =/= j
      then {
        if iter = i
        then _x := x
        else _x := entry.1;
        accum.s_ := accum.s_ + _x;
        accum.c := accum.c * d / (_x * tokens_count);
      }
      else skip;
    } with accum;

  const res = Map.fold(prepare_params, xp, record[ s_ = 0n; c = d; ]);
  ```

- and the second instance of the prepare_params loop

  ```
  function prepare_params(
    var accum        : record[ s_: nat; c: nat; ];
    const entry      : nat * nat)
                     : record[ s_: nat; c: nat; ] is
    block{
      var   _x := 0n;
      const iter = entry.0;
      if iter =/= i
          then {
          _x := entry.1;
          accum.s_ := accum.s_ + _x;
          accum.c := accum.c * d / (_x * tokens_count);
        }
        else skip;
    } with accum;

  const res = Map.fold(prepare_params, xp, record[ s_ = 0n; c = d; ]);
  ```

In these loops, the value `d_P` or the value `c` is computed by repeated multiplication and division during each loop iteration. Therefore, the rounding error in the final value will accumulate and compound over each loop iteration (due to the division).

In each case, the final value of `d_P` or `c` can be understood as a sequence of fractions which are multiplied. However, observe that the numerator and denominator of each fraction in the sequence are independent.

Thus, it is semantically equivalent to multiply each numerator together and each denominator together during the loop body and then perform a single division after the loop end.

Recommendation: We recommend rewriting the loops to defer the divisions in the loop body to a single division operation after the loop end. Since this loop is used a Newton method computation, additional care should be taken to verify via empirical testing that the updated Newton method method converges quickly enough.

Status: This recommendation was adopted in the latest version of the contact.

Liquidity Provider Fee Calculations Can Diverge from Specification

[Severity: Low | Difficulty: Low | Category: Asset Security]

According the fee specification, there are four fee components:

1. liquidity provider fees - for addresses who provide liquidity to the pool
2. referral fees - for addresses that refer users to interact with the pool
3. developer fees - for the developer address
4. staking fees - for addresses that stake a certain token in the pool (only applied when staked value is non-zero)

These fees are configurable (up to a limit) and are charged on all transactions. Finally, in case the total staked value is zero, the staking fees are redirected to liquidity providers.

However, in the case of entrypoint `User_action.Use_dex.Divest_one_coin`, the staking fee is not redirected to liquidity providers when total staked value is 0 and is instead burned.

Recommendation: For consistency, the staking fees should be redirected to liquidity providers when the total staked value is zero.

Status: This issue was fixed in the latest version of the contact.

Pool Creation Cost Calculation Can Diverge from Specification

[Severity: Low | Difficulty: Low | Category: Asset Security, Input Validation]

According to the contract logic, when a user deploys a new dex instance using the factory contract, the contract computes a fee using the following logic (where the fee amounts are stored in variables `to_burn` and `to_factory`):

```
const to_burn = price * burn_rate_f / burn_rate_precision;
const to_factory = abs(price - to_burn);
operations := typed_transfer(
  Tezos.sender,
  Tezos.self_address,
  to_factory,
  Fa2(quipu_token)
) # operations;
operations := typed_transfer(
  Tezos.sender,
  Constants.burn_address,
  to_burn,
  Fa2(quipu_token)
) # operations;
```

where the `typed_transfer` functions send the fees from the user to the factory contract/burn address. The issue is that, when the value `burn_rate_f` is set/reset, it is not checked to be below `burn_rate_precision`. Thus, the subraction in `price - to_burn` may underflow, resulting in an unboundedly large value for both `to_burn` and `to_factory`, instead of the intended where `to_burn` is a fraction of `price`.

Recommendation: Check that `burn_rate_f` is below `burn_rate_precision` when it is set/reset.

Status: This issue was fixed in the latest version of the contact.

Null-valued Storage Map Entries are Not Cleared

[Severity: Low | Difficulty: High | Category: Best practices]

There are several storage maps where a non-existent entry and a null-valued (e.g. zero-valued or empty-set-valued) entry have equivalent semantics. In this case, when a function would reset a map entry to its null value, it is preferable to delete the map entry and reclaim the block storage. This is more important in case the operation semantics permits null-valued operations, in which case griefing attacks can be attempted by submitting null-valued operations to the contract in order to increase deserialization costs. These maps include:

- `storage.ledger` writable via entrypoints `User_action.Use_token.Transfer` and any of the invest/divest liquidity entrypoints
- `storage.allowances` writable via entrypoint `User_action.Use_token.Update_operators`
- `storage.stakers_balance` writable via entrypoint `User_action.Use_dex.Stake`

Notably, all three of these maps can be updated via null-valued operations.

Recommendation: Update functions that write to maps to clear map entries when the final map value is null.

Status: The Madfish Solutions team acknowledged the issue and recommendation by Runtime Verification. They noted that the maps in question are all `big_maps` which are lazily deserialized and for which size changes have only a small effect on overall gas cost. Runtime Verification agrees the effect on the contract gas usage is expected to be only logarithmic in the number of map entries, assuming maps are implemented with a reasonable backing structure like a balanced binary tree, and that this cost is only paid when the maps are accessed and not at contract load time like other data structures.

Duplicate Definition of `get_tokens_from_param`

[Severity: Informative | Difficulty: N/A | Category: Best practices]

Both `partials/factory/helpers.ligo` and `partials/admin/standalone/lambdas.ligo` defined the identical function `get_tokens_from_param`.

Recommendation: Remove one of the definitions.

Status: This issue was fixed in the latest version of the contact.

FA2 Transfers From Undefined LP Tokens are Not Rejected

[Severity: Informative | Difficulty: Low | Category: Input Validation]

When the `User_action.Use_token.Transfer` entrypoint is invoked, zero-valued transfers on undefined LP tokens are not rejected. This behavior is inconsistent with the FA2 standard. Furthermore, combined with the fact that null-valued storage entries are not cleared, this means that attackers or bad implementations may litter the `storage.ledger` map with useless entries.

Recommendation: Update function `iterate_transfer` to reject transfers of LP tokens where the token id is greater than or equal to `storage.pools_count`.

Status: This issue was fixed in the latest version of the contact.

Update Operators Entrypoint Permits Owners as Operators

[Severity: Informative | Difficulty: Low | Category: Input Validation]

Since token owners are always able to send funds under this implementation of the FA2 standard, it is unnecessary for the `User_action.Use_token.Update_operators` entrypoint to allow an owner to mark themselves as an operator. Preventing any unneeded updates also ensures that storage is not wasted.

Recommendation: Reject or filter out updates which assign owners as operators.

Status: This recommendation was adopted in the latest version of the contact.

Minor Documentation Issues

[Severity: Informative | Difficulty: N/A | Category: Best practices]

There were some instances where the code-level comments or associated contract documentation were imprecise:

- The documentation for `set_dev_fee` states:

  > "Dev fee percent couldn't be more than 50%"

  According to the code, it must be less than 50%.

- On documentation website, some entrypoints (including `set_dev_fee`) were described as taking an argument with type `float`. Since the word `float` often indicates an IEEE floating point number, we feel a better choice might be decimal.
- In the code (e.g. here and here), constants were described as multiples of `10e18`, when they were actually multiples of `1e18`.
- In the file `partials/constants.ligo`, some comments next to some binary data incorrectly described how the data was formed.
- In the file `partials/errors.ligo`, the error string for the `not_manager` error was misspelled.
- In the file `partials/dex_core/math.ligo`, the functions `xp_mem` and `xp` (whose return value is a call to `xp_mem`) have different types which are used interchangeably.

Recommendation: Review documentation and code-level comments for clarity.

Status: These issues were fixed in the latest version of the contact.

Minor Constant Definitions Issues

[Severity: Informative | Difficulty: N/A | Category: Best Practices]

In the file `partials/constants.ligo`:

- The constant `dex_func_count`, which defines the final size of the `dex_lambdas` map, was set incorrectly:

  ```
  const dex_func_count: nat = 8n;
  ```

  This constant should be set to be equal to `7n` instead of `8n`. Using an incorrectly larger value enables unreachable code to be assigned to the `dex_lambdas` map which could waste space and gas on-chain.

- The constants `max_fee` and `max_admin_fee` are unused.

Recommendation: Update incorrect constants and removed unused constants.

Status: These issues were fixed in the latest version of the contact.

Code Style Recommendations

[Severity: Informative | Difficulty: N/A | Category: Best Practices]

Here we list some code style suggestions:

1. Most of the contracts core functionality is embedded in lambda functions which are burned into the contract after the contract's intial deployment (to save on gas costs). Due to how this design choice interacts with the type system, each lambda is able to receive any constructor that belongs to its parameter type, even though, by design, it will only ever receive one constructor in its parameter type. To satisfy the type checker, the other unused cases must be handled. As an example of this, we provide the `transfer_ep` function:

   ```
   function transfer_ep(
   const p            : token_action_t;
   var s              : full_storage_t)
                      : full_return_t is
   case p of
   | Transfer(params) -> (Constants.no_operations,
                           List.fold(iterate_transfer, params, s))
   | _                -> (Constants.no_operations, s)
   end
   ```

   The `transfer_ep` function accepts *any* `token_action_t`, but only does meaningful work in case the actual constructor received has the shape `Transfer`. By default, the unused cases all correspond to the identity function on contract storage.

   While this is semantically correct, we believe that throwing an error in these undefined cases, e.g., called `unreachable`, better describes the intent of the programmer.

2. In the `iterate_transfer` function, the pattern shown here:

   ```
   var sender_allowance: set(address) := case s.storage.allowances[sender_key]
   of Some(data) -> data
   |  None -> (set[]: set(address))
   end;
   ```

   can be replaced by the helper function and turned into a constant:

   ```
   const sender_allowance = unwrap_or(storage.allowances[sender_key], set[])
   ```

3. In some cases, pre-condition assertions are deferred past the point where they can be applied. We recommend that pre-condition assertions appear near the top of function definitions. We further recommend that raw `assert`s be replaced with `require` functions which provide meaningful error messages. Some functions where we observed this pattern (details added where relevant):

   - `claim_dev`
   - `ramp_A`

- `add_pool` - (`partials/factory/lambdas.md` version) zero-valued amplification factor is only rejected implicitly because the initial liquidity infusion calls `get_D`

4. In standalone mode, the `fees` parameter of `add_pool` function located in the file `partials/admin/standalone/lambdas.md` is not used. If this parameter is not used, it should be removed.

Recommendation: Consider applying these code style suggestions.

Status: These recommendations were adopted in the latest version of the contact.

Appendix

Table of Entrypoints

For reference, we give a table of entrypoints. See the type/authorization column legend below.

Parameter Name	Called Function	Type	Auth
`Set_dev_function`	`set_function`	F/S	D/A
`Set_admin_function`	`set_function`	F/S	D/A
`Set_dex_function`	`set_function`	F/S	D/A
`Set_token_function`	`set_function`	F/S	D/A
`Use_dev.Set_dev_address`	`set_dev_address`	F/S	D
`Use_dev.Set_dev_fee`	`set_dev_fee`	F/S	D
`Add_pool`	`add_pool`	F/S	U
`Set_init_function`	`set_init_function`	F	D
`Use_factory.Set_burn_rate`	N/A	F	D
`Use_factory.Set_price`	N/A	F	D
`Use_factory.Set_whitelist`	N/A	F	D
`Use_factory.Claim_rewards`	`claim_quipu`	F	D
`Start_dex`	`start_dex_func`	F	U
`Use_admin.Claim_developer`	`claim_dev`	P	D
`Use_admin.Add_rem_managers`	`add_rem_managers`	P	A
`Use_admin.Set_admin`	`set_admin`	P	A
`Use_admin.Ramp_A`	`ramp_A`	P	A
`Use_admin.Stop_ramp_A`	`stop_ramp_A`	P	A
`Use_admin.Set_fees`	`set_fees`	P	A
`Use_admin.Set_default_referral`	`set_default_referral`	P	A
`User_action.Use_dex.Swap`	`swap`	P	U
`User_action.Use_dex.Invest`	`invest_liquidity`	P	U
`User_action.Use_dex.Divest`	`divest_liquidity`	P	U
`User_action.Use_dex.Divest_imbalanced`	`divest_imbalanced`	P	U
`User_action.Use_dex.Divest_one_coin`	`divest_one_coin`	P	U
`User_action.Use_dex.Claim_referral`	`claim_ref`	P	U
`User_action.Use_dex.Stake`	`stake`	P	U
`User_action.Use_token.Transfer`	`transfer_ep`	P	U
`User_action.Use_token.Balance_of`	`get_balance_of`	P	U
`User_action.Use_token.Update_operators`	`update_operators`	P	U
`User_action.Use_token.Update_metadata`	`update_token_metadata`	P	M
`User_action.Use_token.Total_supply`	`total_supply_view`	P	U
`Factory_action.Copy_Dex_function`	N/A	I	F
`Factory_action.Freeze`	N/A	I	F

Abbreviated Name	Effect
`Set_dev_function`	W `.dev_lambdas[i]`
`Set_admin_function`	W `.admin_lambdas[i]`
`Set_dex_function`	W `.dex_lambdas[i]`
`Set_token_function`	W `.token_lambdas[i]`
`Set_dev_address`	W `.storage.dev_store.dev_address`
`Set_dev_fee`	W `.storage.dev_store.dev_fee_f`
`Add_pool`	Deploy new dex contract/add new pool
`Set_init_function`	W `.init_func`
`Set_burn_rate`	W`.storage.burn_rate_f`
`Set_price`	W`.storage.init_price`
`Set_whitelist`	W`.storage.whitelist`
`Claim_rewards`	W `.storage.quipu_rewards = 0`; Send value to developer
`Start_dex`	Start factory deployed dex contract
`Claim_developer`	Subtract value from `.storage.dev_rewards[i]`; Send to developer
`Add_rem_managers`	W `.storage.managers`
`Set_admin`	W `.storage.admin`
`Ramp_A`	W `.storage.{ initial_A_{f, time}, future_A_{f, time}`
`Stop_ramp_A`	W `.storage.{ initial_A_{f, time}, future_A_{f, time}`
`Set_fees`	W `.storage.pools[i].fee`
`Set_default_referral`	W `.storage.default_referral`
`Swap`	Swap one token for another
`Invest`	Invest liquidity into a dex
`Divest`	Divest liquidity from a dex in equal proportion
`Divest_imbalanced`	Divest liquidity from a dex in unequal proportion
`Divest_one_coin`	Divest liquidity into only one coin
`Claim_referral`	Subtract value from `.storage.referral_rewards[i]`; Send to referrer
`Stake`	Stake or unstake QIUPU token and claim staking rewards
`Transfer`	Send LP token
`Balance_of`	Get balance of LP token
`Update_operators`	Update operators for a LP token holder's address
`Update_metadata`	Update metadata for a given FA2 contract
`Total_supply`	Print current total supply for LP token
`Copy_Dex_function`	W `.storage.dex_lambdas`
`Freeze`	W `.storage.started`

Table of Entrypoints Legend

Type Legend

1. F - Factory Contract
2. S - Standalone Dex Contract
3. P - Pool (Standalone Dex Pools and Factory Dex Pools)
4. I - Factory Dex Contract Instances

Authorization Legend

1. D - developer address (can call developer entrypoints)
2. F - factory address (can all factory actions)
3. A - admin address (can call admin entrypoints)
4. M - one of the manager addresses (can call `Update_metadata`)
5. U - user addresses (can call `User_action` entrypoints)

Effect Legend

We abbreviate effects which update a particular subset of storage fields. We use the notation:

1. To update a standard storage field:

   ```
   W .storage_field
   ```

2. To update a nested storage field, iterate the (`.`) operator:

   ```
   W .field.nested_field
   ```

3. To update a map-like storage field, use the `[]` notation:

   ```
   W .field[i]
   ```

   to refer the the `i`th field in the map `.field`.

4. For multiple nested fields underneath a single field with a shared prefix, use:

   ```
   W .field.{ prefix_{ nested_field1, nested_field2 } }
   ```

   where `.field` has two nested fields:

   - `prefix_nested_field1`
   - `prefix_nested_field2`

Newton Method Analysis

We analyze the Newton method loop body used in swap/single coin divestment computation:

$$y' = \frac{y^2 + c}{2y + b - D}$$

where the loop continues to iterate until it converges, i.e., until $|y' - y| < \delta$ for some $\delta > 0$ and we have the abbreviations:

- $c = \frac{D^{n+1} * A_{precision}}{A * n * n^n * \prod X/\{x_j\}}$;
- $b = \sum X/\{x_j\} + \frac{D * A_{precision}}{A * n}$.

and where all constants are non-negative.

This loop is derived from applying the Newton method to the stableswap invariant rearranged to be an a function $f(y)$. We now give the definition of the function $f(y)$ below:

$$f(y) = y^2 + y * \left(\sum X/\{x_j\} + \frac{D * A_{precision}}{A * n} - D\right) - \frac{D^{n+1} * A_{precision}}{A * n * n^n * \prod X/\{x_j\}}$$

where:

- Each token balance in the AMM is referred to by some x_i where $0 < x_i$
- For each token balance x_i, we have $x_i \in X$
- We require $0 < A$
- We require $0 < A_{precision}$
- We require $2 \leq n \leq 4$
- We require $0 < D$
- x_j is the token balance of the output token (i.e. the token that the swapper/liquidity provider is purchasing from the AMM)
- $X/\{x_j\}$ is the set of tokens balances in the dex with j^{th} token removed
- for ease of reading, the dependent variable y is a renaming of x_j

We want to use the Newton method to find the unique non-negative zero of this function, i.e., when $f(y) = 0$ such that $0 < y$. Note that, using Descartes' rule of signs, we can prove that such a zero exists and that it is unique.

We want to prove that Newton's method will ***over-approximate*** *this unique, positive root.*

To do so, we first derive the Newton method loop body from the Newton formula, that is:

$$y' = y - \frac{f(y)}{f'(y)}$$

To compute this formula, we need the derivative of f:

$$f'(y) = 2y + \sum X/\{x_j\} + \frac{D * A_{precision}}{A * n} - D$$

Using the abbreviations c and b from above, we can derive the Newton method loop body:

$$\begin{aligned}
y - \frac{f(y)}{f'(y)} &= y - \frac{y^2 + (b-D) * y - c}{2y + b - D} \\
&= \frac{y * (2y + b - D) - (y^2 + (b-D) * y - c)}{2y + b - D} \\
&= \frac{2y^2 + (b-D) * y - y^2 - (b-D) * y + c}{2y + b - D} \\
&= \frac{y^2 + c}{2y + b - D}
\end{aligned}$$

Now we can prove that the Newton method will over-approximate the unique, positive root because $f(y)$ is:

- *convex* (all quadratics with a positive leading coefficient are convex, since the second derivative is non-negative everywhere),
- continuous (since $n > 0$, $A > 0$, each $x_i > 0$ for $x_i \in X$, and $D > 0$), and
- the derivative is non-zero everywhere except the global minimum.

Recall that Newton's method uses a curve's tangent lines to gradually approximate its roots. That is, the Newton formula approximates the root of a curve f by:

1. computing the tangent to f at some point y;
2. setting y' to be the unique zero of the tangent line at y (well-defined when $f'(y) \neq 0$).

Using real arithmetic, we can prove that applying Newton's method using real arithmetic to a continuous, convex curve to any point on the right of the global minimum will *over-approximate* the rightmost root, since the tangent line to any point on the curve *must lie on or below* the curve due to convexity.

Furthermore, we can prove that, in theory, the contract will always choose a starting point for the Newton method to the right of the global minimum. Even if this fails in practice (e.g. due to rounding), the contract will revert due to either a subtraction underflow (when $2y + b < D$) or else a division by zero (when $2y + b = D$).

By manipulating rounding error in the loop body, we can *prove* that our integer implementation of the Newton method will *over-approximate* the real arithmetic implementation of the Newton method which will *over-approximate* the root.