

Ref.Finance Security Audit

First Release

March 2022

Auditor: Dr J. Rousselot

Executive Summary

Ref Finance is the most widely used DEX on NEAR protocol

It is written in the Rust programming language.

The audit covers the Ref Finance codebase available at

<https://github.com/ref-finance/ref-contracts>

commit `3c04fd20767ad7f1c383deee8e0a2b5ab47fbc18` dated 17 December 2021 on branch `main`.

The code is organized in two projects: `ref-exchange` and `ref-farming`.

To the best of our knowledge, we did not find any critical security issue with the codebase in the version audited.

Issues

We made the following observations during our audit. More details and recommendations can be found in the respective sections.

Each issue risk severity is estimated following the OWASP risk rating methodology. An overall risk severity is made based on both the likelihood of the issue being exploited and the technical/business/financial impact of the issue.

Issue	Risk Severity	Explanation
1.4.1 Version pinning	High Likelihood: 5 Impact: 7	It is theoretically possible for an attacker to compromise a third party dependency and inject an attack through it. Recommendation: add version pinning.
1.4.3 Reproducible builds	High Likelihood: 5 Impact: 9	Currently, it is not possible to prove that the DEX code deployed on chain matches the source code on github. Recommendations: improve docker builds and prioritize reproducible builds.
1.6.1.3 Always use <code>assert_one_yocto</code> in <code>owner.rs</code>	High Likelihood: 5 Impact: 7	All functions restricted to owner and/or guardians should use the <code>assert_one_yocto</code> . This prevents these user roles from delegating access to sub-accounts.
1.5.1 Unit test coverage	Low Likelihood: 2 Impact: 7	Some security critical smart contract functions could be covered by unit testing.
1.5.2 Unit test framework	Low	The NEAR unit testing framework relies on a

	Likelihood: 2 Impact: 7	simulator rather than actual blockchain node code.
1.6.2.3/1.6.2.4 the code is slightly too complex as only one action is supported at the moment.	Comment Likelihood: 2 Impact: 2	Only the SWAP action is currently implemented. If no other actions are planned, the code could be simplified.
1.6.2.5 Non fully standardized error codes	Comment Likelihood: 2 Impact: 2	Instead of using a string for the error, it is better coding practice to return a specific item from the Errors enum (ERR_NO_POOL).
1.6.2.5 Temporary double spend within an atomic tx	Comment Likelihood: 1 Impact: 7	When adding liquidity to a pool, very temporarily more tokens exist in the system during the function call. As transactions are atomic and not parallelized there is no security risk.
1.6.3.1 always use the same numeric constant NUM_TOKENS	Comment Likelihood: 1 Impact: 3	The constant NUM_TOKENS can be used to check whether function arguments have the correct size. Using this coding style is clearer than referencing other objects which have previously been checked with NUM_TOKENS such as token_account_ids.len()
1.6.3.9 refactor assert to compare object length with NUM_TOKENS into a function check_num_tokens()	Comment Likelihood: 1 Impact: 3	Input validation of vectors can be further refactored by introducing an internal function check_num_tokens_length(Vec<>).

Disclaimer

The auditor and the auditing company made their best efforts to identify any security issue related to the codebase under audit.

They cannot be held responsible for any remaining security vulnerability in the software product or for any financial loss that the usage of the software product may cause to anyone.

A security audit is not financial advice.

1 REF-Exchange	5
1.1 User Roles	5
1.2 Pools	5
1.3 Tokens	6
1.3.1 Fee on Transfer	6
1.3.2 Rebasing Tokens	6
1.4 Software Architecture	7
1.4.1 Version pinning	8
1.4.2 Docker builds	8
1.4.3 Reproducible builds	9
1.5 Unit Tests	9
1.5.1 Unit test coverage	9
1.5.2 NEAR Simulation framework	11
1.6 Smart Contracts	11
1.6.1 owner.rs	11
1.6.2 lib.rs	11
1.6.3 simple_pool.rs	12
1.7 Sequences	13
1.7.1 Creation of a new pool	13
1.7.2 Swap	14
1.7.3 Providing liquidity	14
1.7.4 Withdrawing liquidity	14

1 REF-Exchange

REF-Exchange is a decentralized exchange running on the NEAR blockchain.

The following sections respectively describe the different user roles within REF, the functioning of the pools, the software architecture of the DEX, a review of the unit tests and the audit of the smart contract codebase itself.

1.1 User Roles

A DEX user can be either a *liquidity provider (LP)* or a *trader*.

Any REF user can be a LP or a trader.

Besides LP and traders, there are two other user roles in the system: *owner* and *guardians*.

As a *liquidity provider*, any NEAR user can:

- create a new trading pool using the white listed tokens,
- provide liquidity to an already existing ref-exchange trading pool,
- withdraw liquidity from an existing pool.

As a *trader*, any NEAR user can also:

- make a trade on a pool.

The *owner* can:

- add and remove guardians,
- create new pools,
- upgrade the smart contract,
- change the owner,
- change the REF Exchange state (running or paused),
- set the fees,
- do most/all actions allowed for guardians.

Guardians can:

- add/remove whitelisted tokens,
- set the REF Exchange state to paused,
- set the fees.

1.2 Pools

The trading pool implements the same pricing mechanism as uniswapV1 and V2: $X*Y=K$.

As described in the UniSwapV2 docs: *Where X and Y represent the respective reserve balances of two ERC-20 tokens, and "K" represents the product of the reserves.*

There are two challenges with such pools:

1. They require a large initial deposit by the pool creator.
2. Impermanent loss to LP.

In practice, the large initial deposit is difficult to quantify. The REF team made the design choice to let pools trade with any amount of tokens. If we consider very valuable tokens, it is realistic for users to trade small amounts of token A against small amounts of token B.

The REF frontend implements several features to help the user make informed decisions.

1.3 Tokens

Besides, there are also several possible trading errors associated with $X*Y=K$, which depend on the algorithmic behavior of the tokens being traded. With ETH / UniswapV2, the most common ones are *fee on transfer* tokens, and *rebasing* tokens.

These issues are caused by the token smart contracts and their operators themselves, rather than the DEX codebase. However, these issues can interfere with the proper operation of a DEX.

More info on the UniswapV2 issues can be found at the following link:

<https://docs.uniswap.org/protocol/V2/reference/smart-contracts/common-errors#:~:text=The%20Uniswap%20constant%20product%20formula,the%20%E2%80%9CK%E2%80%9D%20error%20refers.>

1.3.1 Fee on Transfer

Fee on transfer is caused by some ERC20 tokens charging a fee on each transfer.

On Ethereum, this can break the pool as the equation $X*Y = K$ is not true anymore.

With REF / NEAR, the user first deposits tokens to the exchange before moving them into the pool. Tokens moved to/from the pool are not moved from the token's point of view (see section 1.7 for details). Therefore the Ethereum ERC20 Fee on Transfer should not be an issue with the REF pool operation. At the time of writing no such tokens seem to exist on NEAR yet.

It would be useful to implement a unit test to confirm this analysis.

1.3.2 Rebasing Tokens

Rebasing tokens are caused by the token administrators / issuers issuing more tokens, or alternatively burning tokens.

- A) Consider a token A with 100 units in circulation. If the administrators suddenly issue 900 more tokens and distribute them equally among token holders, all things being equal, the value of this token is likely going down by a factor 10.
- B) Conversely, if the admin of token A with 100 units in circulation burns 90 of the tokens, equally from each token holder, then the value of each token suddenly climbs by a factor 10, all things being equal.

In scenario A, the DEX would hold for instance 10 units. All 10 tokens have been allocated to one pool. Then, token A admins issue and distribute 90 more tokens. These new tokens have

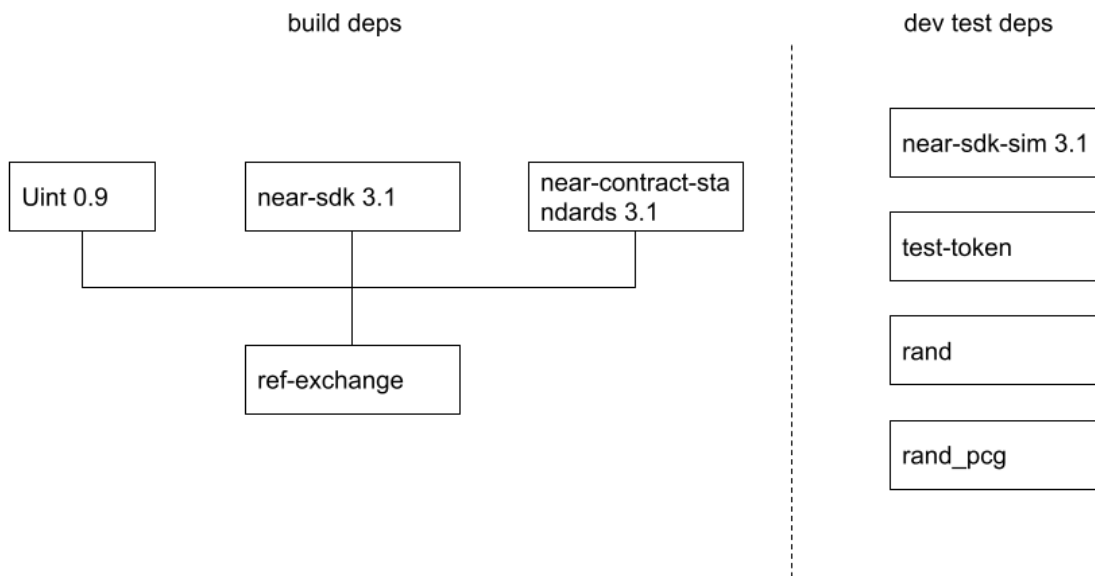
not been allocated to any pool. So the pool balance remains the same. These tokens are not owned by anyone. Arbitrage players will rebalance the pool, adding more A and withdrawing Y. In scenario B, the DEX would hold 10 units. All 10 tokens have been allocated to one pool. After burning 90% of the tokens, only 1 is left from the token's point of view. However, the DEX does not know about this. In the pool, there are still 10 (assuming all allocated to the pool). When users try to withdraw these tokens from the DEX, this will fail as from the token's contract point of view, there are 10 times fewer tokens than the DEX assumes.

1.4 Software Architecture

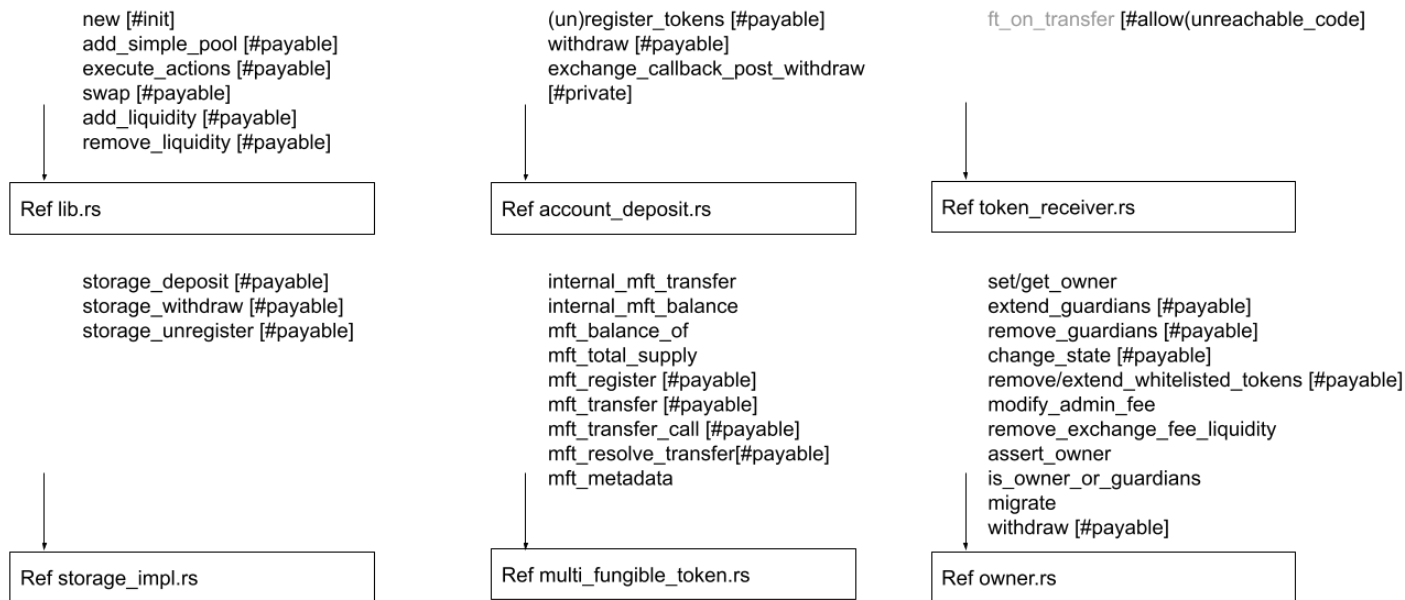
The ref-exchange codebase is organized as follows.

1. src holds the smart contracts themselves,
2. tests directory holds all the tests.
3. tests/fuzzy holds some randomized tests.
4. ../test-token an external directory holds the source code for a test token.

The figure below represents all package dependencies for both the production build target and for the development build target. We can see that few packages are used. This is a good security practice as it minimizes third party risks.



The next figure shows the list of functions that can be called by users of the smart contract.



Not shown: legacy.rs and views.rs

1.4.1 Version pinning

The build system uses Cargo, the Rust package manager. Some dependencies in the Cargo manifest file are insufficiently specified. For instance, package uint is set at version 0.9. This lets the build system silently download newer patch versions (0.9.2, 0.9.3) than the one intended by the developer (0.9.1). As the uint package maintainers released a breaking change in 0.9.2, the code becomes impossible to build (as it requires a more recent version of rust). Requesting package versions including the patch number, or using Cargo lock files would make sure all developers build using the exact same libraries.

The Rust specification explains that patch number should not introduce breaking changes:

<https://doc.rust-lang.org/cargo/reference/manifest.html#the-version-field>

The uint library upgraded Rust edition from 2018 to 2021 here:

<https://github.com/paritytech/parity-common/commit/c3ef97d403bf30fb4ff35ce9ac74b60c3a0c8f42#diff-fef31bc61f5ea5a6eaf423dc2ebb52c192debb61a579da44d9b58768a27f2f7>

Which requires rustc 1.56.1:

<https://github.com/paritytech/parity-common/pull/601>

1.4.2 Docker builds

Docker is an important tool to standardize the build system and to make sure developers work on the same software version.

We recommend that REF fixes the Docker build scripts at least for the intel/amd64 Linux platform and introduce a Dockerfile in the repository instead of relying on Makefiles / shell scripts.

The table below summarizes our attempts to build the software on various platforms.

	amd64 Ubuntu 20.04 build_local.sh	amd64 Ubuntu 20.04 build_docker.sh	Apple Silicon MacOS 12.2 build_local.sh	Apple Silicon MacOS 12.2 build_docker.sh
ref-exchange contracts	OK	breaks	OK	breaks
ref-farming	OK	breaks	OK	breaks

1.4.3 Reproducible builds

We found a major issue around the system on-chain deployment. Despite our best effort, we could not reproduce the WASM binary used by REF Finance for on-chain deployment. We recommend REF to prioritize reproducible builds. This is partly a wider NEAR ecosystem issue, but immediate actions related to dependency packages pinning can be taken now, as well as using docker build automation. As of writing, we have no way of proving that the code deployed on chain is the same as the code being audited.

The ultimate goal of reproducible builds is to generate the exact same binary file (verified with a hash) on various platforms: amd64 and arm architectures, Linux and MacOS operating systems. We recommend that REF implements version pinning, fixes Docker builds and works on reproducible builds.

1.5 Unit Tests

We reviewed several Rust tools to generate a unit test code coverage report: GRCOV, LCOV and Cargo Tarpaulin.

Cargo Tarpaulin is a framework that generates html reports and we recommend its use.

More details can be found in the Tarpaulin report.

For reference, tarpaulin can be run as follows:

```
Requires: Linux x64
apt install pkg-config libssl-dev
cargo install tarpaulin-cargo
cargo tarpaulin
```

1.5.1 Unit test coverage

The figure below from the tarpaulin report shows that the files admin_fee.rs, lib.rs, action.rs, simple_pool.rs, utils.rs are well covered (85 to 100% test coverage).
















We also observe that the files legacy.rs, owner.rs, storage_impl.rs, token_receiver.rs and views.rs have low or minimal coverage. While some code such as view functions may be low risk, it would be good practice to increase the coverage for most of the code. It is especially important for security sensitive functions such as the code in owner.rs.

This effort would prove programmatically (within the tests scenarios considered) that the security mechanisms are correctly implemented. It would also prevent any future mistake from degrading this security level, either by changes to the REF smart contract themselves or from silent dependency package upgrades through the Cargo package manager. Currently REF does not use a Cargo lock file, and does not use pin package versions in the Cargo.toml file directly. Third party dependencies can thus introduce changes, although luckily few dependencies are used by the codebase at the moment.

To summarize, we recommend to increase unit tests code coverage for the following modules:

- account_deposit.rs
- owner.rs (set/get owner/guardians, change_state, modify_admin_fee, remove_exchange_fee_liquidity)
- pool.rs
- token_receiver.s (especially the instant swap does not seem to be covered)

Ideally, all functions in the list of entry points in the previous section should be well tested.

 stable_swap	507 / 599 (84.64%)
 account_deposit.rs	110 / 164 (67.07%)
 action.rs	6 / 7 (85.71%)
 admin_fee.rs	4 / 4 (100.00%)
 errors.rs	0 / 0
 legacy.rs	0 / 5 (0.00%)
 lib.rs	443 / 500 (88.60%)
 multi_fungible_token.rs	55 / 101 (54.46%)
 owner.rs	10 / 77 (12.99%)
 pool.rs	31 / 73 (42.47%)
 simple_pool.rs	176 / 186 (94.62%)
 storage_impl.rs	21 / 49 (42.86%)
 token_receiver.rs	6 / 27 (22.22%)
 utils.rs	19 / 19 (100.00%)
 views.rs	19 / 85 (22.35%)

Note that because of the current state of Rust unit tests coverage tools, it is possible that tarpaulin report is not entirely accurate. Some new features exist in the experimental branch of

Rust toolchain but the NEAR framework is not compatible with them yet (as they require wasm 2 and NEAR uses wasm 1.x).

1.5.2 NEAR Simulation framework

All unit tests from REF use the so-called NEAR simulation framework. This is the most widely used and most developed tool for unit testing within REF.

However, NEAR has developed the near-sandbox to run a local blockchain for testing.

This approach has several advantages over the NEAR simulator:

- More realistic as uses same codebase as production NEAR nodes,
- Gas cost estimations,
- Ability to write and run tests in NodeJS,
- As tests are run through RPC they can also be run on testnet and mainnet.

At the time of writing, it seems that NEAR Sandbox is not ready for production yet.

We recommend that at some point in the future, test development migrate over Sandbox.

1.6 Smart Contracts

1.6.1 owner.rs

We found no specific issue in this module. The code is clean and easy to understand and to audit. Improving unit testing would further improve the code quality.

1. **set_owner:**
This is a critical function. If an incorrect new owner is set, there is no way to recover from this and all fees will be lost.
2. **change_state:**
Only the owner can resume the contract after it has been paused.
3. **assert_one_yocto:**
All functions in this module will benefit from using `assert_one_yocto()` to prevent owner and guardians from delegating their capabilities and increasing attack surface (more keys -> more possibility that a key gets stolen).

1.6.2 lib.rs

This is the core module of ref-exchange.

Contract

1. **init/new**
2. **add_simple_pool:**
by design, the system lets users create multiple pools with the same token pair. This enables competition on fees but can reduce liquidity.
The total fee set by the pool creator is verified to be under a constant maximum value.

3. **execute_actions:**
batched atomic DEX actions execution for any account. Some input validation. At the moment only one action (Action::Swap) is supported. The code will reject any other action type as enum Action only has one enum. We recommend filtering either explicitly at the entry_point the Action values, and/or implementing a default error handler for internal_execute_action. The code can be made more explicit and robust by filtering out immediately unimplemented actions. Incorrect poolId is filtered and will stop tx.
4. **swap:**
Let a user execute multiple SwapActions in a single tx. Contract mode is checked. The difference between the functions swap and execute_actions is not clear. If no other actions are under development, it may be better to simplify the codebase and keep only one entry point.
5. **add_liquidity:**
This function adds liquidity to a pool in proportion to the pool ratio. "ERR_NO_POOL" should be from the errors enum .
Ideally, we always want to subtract amounts before adding amounts when operating fund transfers. Here we add tokens to the pool before removing them from the user's account. As this is all internal accounting within the DEX smart contract / account, and contained within a single atomic transaction, it is very unlikely to be a risk. Still, for a very short time we create too many tokens in the system (tokens are both in the user's deposit account and in the pool).
6. **remove_liquidity:**
This function removes some or all of the liquidity previously deposited by a LP.

1.6.3 simple_pool.rs

1. **new**
Use NUM_TOKENS for initialization of amounts, volumes, instead of token_account_ids.len()
2. **share_register**
3. **share_transfer**
4. **share_balance_of**
5. **share_total_balance**
6. **tokens**
7. **add_liquidity**
8. **mint_shares**
9. **remove_liquidity**
Refactor assert_eq!(...) wrong token count into a function check_token_count(Vec<T>); always compare directly with NUM_TOKENS (rather than self.token_accounts_ids.len())
ERR_WRONG_TOKEN_COUNT, ERR_SHOULD_HAVE_2_TOKENS
Add ERR to Errors enum

10. token_index

Simple function but important for security. As input tokens are validated through this call. Add ERR to Errors enum

11. internal_get_return

Critical function for trade estimation. Returns the estimated amount V of token B for input amount of token A. The formulas used in the codebase are an optimization to improve arithmetic accuracy. The following computations below verify the formulas implemented in the codebase.

We have:

$$K = \text{amount_in} * (\text{FEE_DIV} - \text{total_fee}) = \text{amount_in} * \text{FEE_DIV} * (1 - \text{total_fee} / \text{FEE_DIV}) \\ = \text{amount_in} * \text{FEE_DIV} - \text{amount_in} * \text{total_fee}$$

$$V = K * \text{out_balance} / (\text{FEE_DIV} * \text{in_balance} + K) \\ = [\text{amount_in} * \text{FEE_DIV} * \text{out_balance} - \text{amount_in} * \text{total_fee} * \text{out_balance}] / [\text{FEE_DIV} * \text{in_balance} + \text{amount_in} * \text{FEE_DIV} - \text{amount_in} * \text{total_fee}] \\ = \text{amount_in} * \text{out_balance} (1 - \text{total_fee} / \text{FEE_DIV}) / [\text{in_balance} + \text{amount_in} (1 - \text{total_fee} / \text{FEE_DIV})] \\ = \text{amount_in} * \text{out_balance} / (\text{in_balance} + \text{amount_in})$$

12. get_return

wrapper for internal_get_return.

This function validates that both input and output tokens requested correspond to the pool and return the result of internal_get_return as a Balance object.

13. get_fee

Getter function for total_fee

14. get_volumes

Getter function for volume statistics

15. swap

Performs the swap operation based on the amount from internal_get_return.

Verifies that the pool invariant is respected.

Mint and distributes shares as needed.

Finally, update volume statistics.

1.7 Sequences

This section describes how the product features described in section 1.1 are implemented function call by function call.

It is possible to execute these features using alternate / simplified flows if the user already has some internal balance within the DEX.

1.7.1 Creation of a new pool

1. The owner or a guardian whitelists a new guardian G through extend_guardians.
2. The new guardian G whitelists a new token T through extend_whitelisted_tokens.*
3. A user U (possibly same as G) creates a new pool add_simple_pool.

4. User U deposits some amount of token T to the contract account and NEAR calls `ft_on_transfer` and `internal_deposit`.
5. User U deposits some amount of previously whitelisted token W in the same manner as in the previous step.
6. User U adds liquidity with token T, NEAR or token T, token W to the pool using `add_liquidity`.
7. If the pool has enough liquidity, it is possible to trade.

(*) Step 2 is optional. It is technically possible to create a pool of non whitelisted tokens. However, the REF frontend will filter it out from end users.

1.7.2 Swap

Once a pool exists and has enough liquidity, the following sequence is possible:

1. The user deposits an amount of token T to the DEX.
2. The user calls function `swap` with an action of type `SWAP` (only action supported so far) to convert some of his token T balance to the pool pair asset. If the amount is above the minimum amount, the user balance in the pool is updated accordingly.

1.7.3 Providing liquidity

A user can contribute liquidity to a pool as follows:

1. User U deposits some amount of token T to the contract account and NEAR calls `ft_on_transfer` and `internal_deposit`.
2. User U deposits some amount of previously whitelisted token W in the same manner as in the previous step.
3. `add_liquidity` transfers the liquidity from the user's internal account within the DEX to the actual pool. The user is credited shares in the pool.

1.7.4 Withdrawing liquidity

If the user already contributed to a pool liquidity, it is possible to withdraw it using the following function call:

1. `remove_liquidity` will reduce the pool liquidity accordingly, remove the shares from the user's balance and credit its internal token balance.