Security Audit Report

Loopring Protocol (V3)

SECBIT

Nov 15, 2019

1. Introduction

Loopring Protocol (V3) is a decentralized exchange protocol for blockchains. SECBIT Labs conducted an audit from Aug 15th to Nov 15th, 2019, including an analysis of **Smart Contract** and **Circuit Code** in 3 areas: **code bugs**, **logic flaws** and **risk assessment**. The audit results show that Loopring Protocol (V3) has no critical security risks, and the SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Implementation	4.3.1 Definitions and implementations of `getLRCFeeStats()` interface are inconsistent	Info	Updated
Implementation	4.3.2 Assembly code in `SignatureBasedAddressWhitelist` does not work	Low	Updated
Mechanism Design	4.3.3 Operator influence on the choice of maker/taker orders	Low	Discussed
Best Practice	4.3.4 Unchecked return values in use of `transferTokens()`	Info	To Be Discussed
Implementation	4.3.5 Unclear intention of `transferDeposit()` function	Medium	To Be Discussed
Protocol Design	4.3.6 Consider about overflow issues of timestamp in `Block`	Low	To Be Discussed
Circuit Code	4.3.7 `TransformRingSettlementDataGadget` could be optimized	Info	To Be Discussed
Documentation	4.3.8 Wrong comment in `TakerMakerMatchingGadget`	Info	To Be Discussed
Circuit Code	4.3.9 Call logic of `generateKeyPair()` is impractical	Low	Updated
Circuit Code	4.3.10 Redundant fields in the `OffchainWithdrawalBlock` class	Low	To Be Discussed

2. Project Information

This part describes the basic information and code structure.

2.1 Basic information

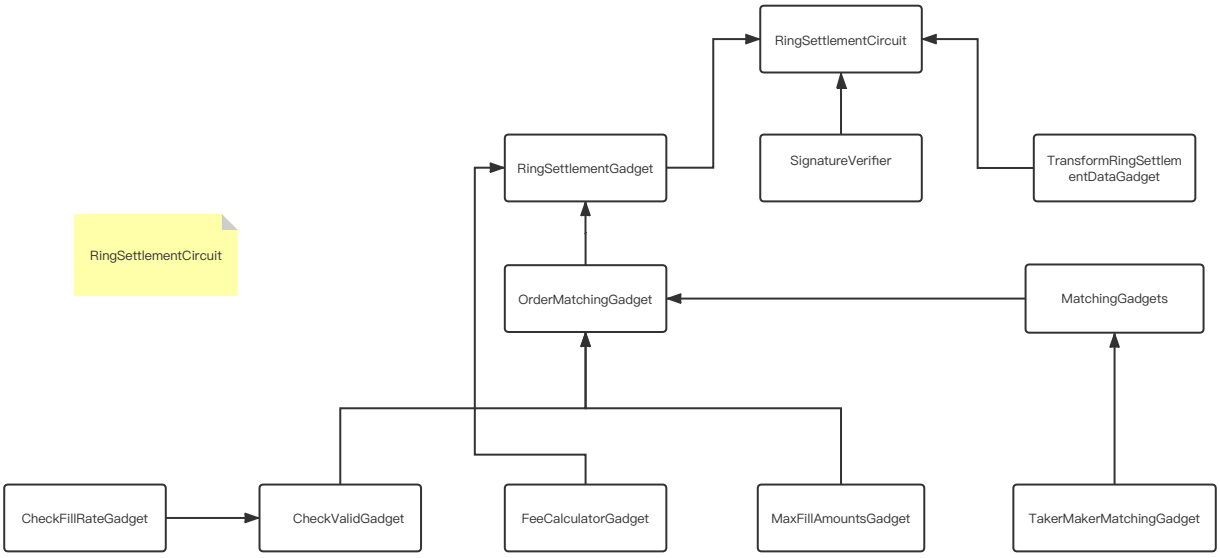
The basic information about the Loopring Protocol (V3) is shown below:

- **Project website**
  - https://loopring.org/#/protocol
- **Design details of the protocol**
  - https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/DESIGN.md
  - https://github.com/Loopring/whitepaper
- **Smart contract code**
  - https://github.com/Loopring/protocols/tree/3.0-beta3/packages/loopring_v3
  - Commit `10100aa616223439516c48f2c76ef386e8f996ff`
- **Circuit code**
  - https://github.com/Loopring/protocol3-circuits/tree/3.0-beta3
  - Commit `18d853d67aed649aeb2a0487870c6a37773d0d64`

2.2 Circuit List

The following content shows the circuits included in the Loopring Protocol (V3) project:

Circuit	Description
DepositCircuit	On-chain deposit
OffchainWithdrawalCircuit	Off-chain withdrawal
OnchainWithdrawalCircuit	Off-chain withdrawal
OrderCancellationCircuit	Order cancellation
RingSettlementCircuit	Ring settlement for orders



(Internal structure of RingSettlementCircuit)

Each circuit corresponds to a specific type of business logic of DEX. Computations and proof generations happen off the chain. Data and proofs are submitted to the contract to get verified. The figure above shows the internal structure of the `RingSettlementCircuit`. See the appendix for a more detailed analysis of circuits.

2.3 Contract List

The following content shows the main contracts included in the Loopring Protocol (V3) project:

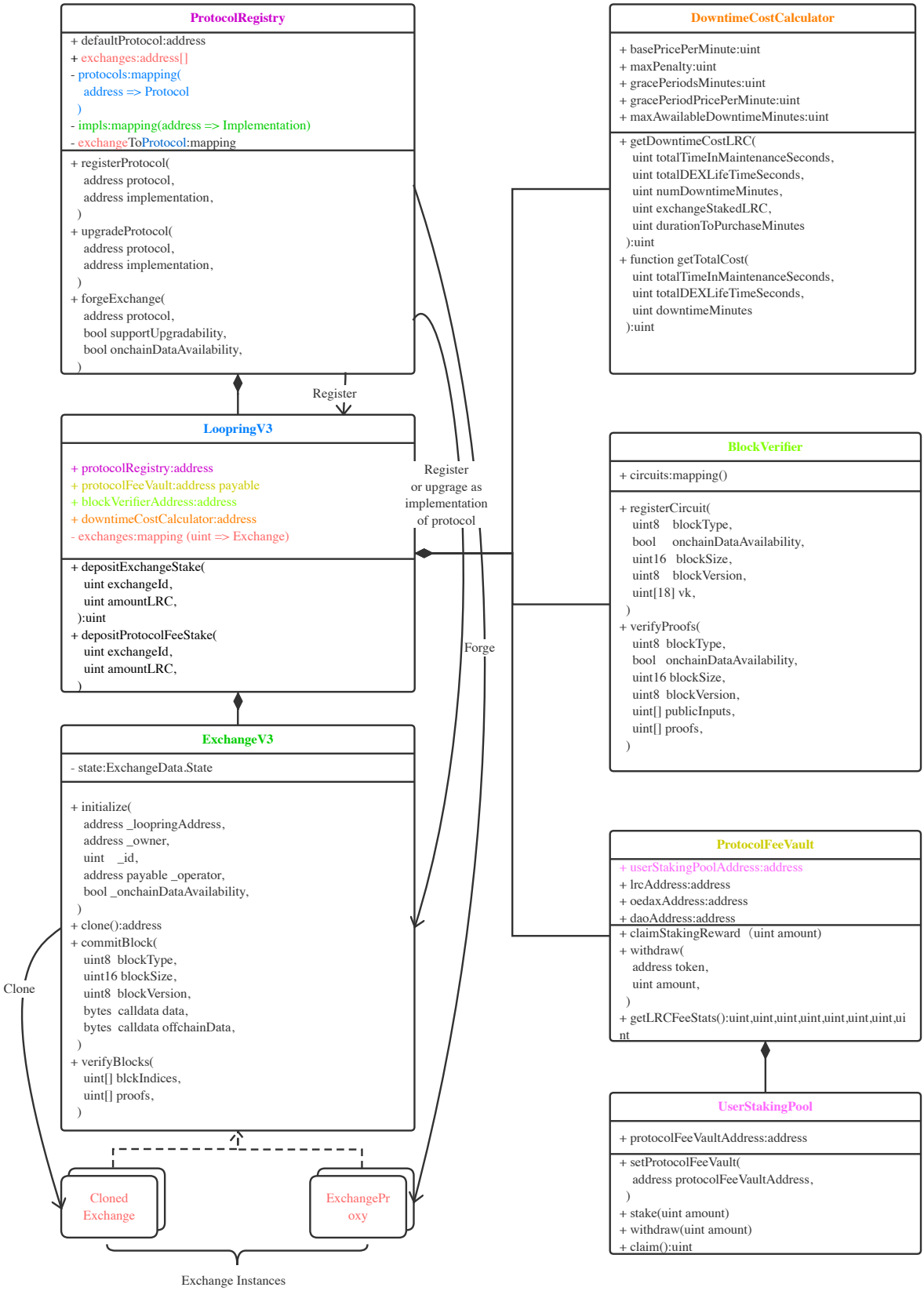
Contract	Description
ProtocolRegistry	Portal of protocol registration and management
LoopringV3	Loopring core contract
ExchangeV3	Exchange core contract
ExchangeProxy	Provide upgradability for exchange by proxy mechanism
BlockVerifier	Verifier contract for circuit proofs
ProtocolFeeVault	Manage the distribution of protocol fees
UserStakingPool	Handling users staking related logic
DowntimeCostCalculator	Calculate and manage downtime cost

SignatureBasedAddressWhitelist	Signature-based user whitelist contract
LzDecompressor	Lz decompression contract

The picture below shows the dependencies and call logic of core smart contracts. Implementations can be divided into 3 layers:

`ProtocolRegistry -> Protocol -> Exchange`

The `ProtocolRegistry` is the entry of protocol registration and management, for registering and managing the implementation of the protocol. The `Protocol` is the protocol implementation, which is used to manage the sub-exchanges of itself. The `Exchange` is the implementation of the specific exchange. One can create and run multiple instances of it.



(Code Structure of Loopring V3 Smart Contract)

3. Code Analysis

This part describes details of code assessment, including 2 items: "role classification" and "functional analysis".

3.1 Role Classification

There are several key roles in the protocol, namely Registry Owner, Protocol Owner, Exchange Owner, Exchange Operator, Exchange User, and Normal User.

- Registry Owner
  - Description

    Administrator for protocol registry contract
  - Authority
    - Register or upgrade protocol
    - Disable or enable protocol
    - Set the default protocol
  - Method of Authorization
    - The creator of the `ProtocolRegistry` contract automatically becomes `owner`
    - Authorized by transferring the ownership of the contract
- Protocol Owner
  - Description

    Protocol administrator
  - Authority
    - Update settings of protocol
    - Update fee settings
  - Method of Authorization
    - `Protocol` contract creator automatically becomes `owner`
    - Authorized by transferring the ownership of the contract
- Exchange Owner
  - Description

    Exchange administrator
  - Authority
    - Register Tokens
    - Turn on/off the deposit function of a specific token

        - Withdraw the assets staked by the exchange owner
        - Handling assets that were mistakenly transferred to the exchange address
        - Set exchange operator
        - Set fees
        - Set up a whitelist contract
        - Turn on/off maintenance mode
        - Shutdown the exchange after properly processed the users' assets
    - Method of Authorization
        - `Exchange` contract creator automatically becomes `owner`
        - Authorized by transferring the ownership of the contract
- Exchange Operator
    - Description

      The operator who is responsible for updating the exchange states
    - Authority
        - `commitBlock()`
        - `verifyBlocks()`
        - `revertBlock()`
        - `withdrawBlockFee()`
    - Method of Authorization

      Set by Exchange Owner
- Exchange User
    - Description

      Users who created accounts in the exchange
    - Authority
        - Account registration and updating
        - Deposit and withdraw
    - Method of Authorization

      User registered through the exchange contract
- Normal User
    - Description

      Normal Ethereum account
    - Authority

      Execute other operations allowed by the contract
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The Loopring V3 adopts the design approach similar to zk-rollup. As known, zk-rollup uses a Merkle tree to store account status. The smart contract only needs to save the Merkle root. The child node of the tree is a map of `AccountID => (pubkey, balance)`. Here, zk-rollup is most simplified into transfers of a single type of currency between accounts.

Decentralized exchanges like Loopring need to maintain more states, such as balances and order history of each currency. Loopring uses a sub-Merkle tree to maintain account, balance, and trade history. It also uses the quad-Merkle tree structure and Poseidon hash to future improve performance.

Loopring V3 stores all states of a single sub-exchange into a Merkle tree, in which the exchange contract only saves the root of it. The rest of the states is saved and computed off the chain. The `calldata` is used to store critical data at the same time. So the data availability is guaranteed when reducing the cost of on-chain storage.

The key functions are divided into the following items:

- `commitBlock()`

  The exchange operator is responsible for submitting blocks to the contract. Note the distinction with block concept in the blockchain. Here the block means a virtual block in the smart contract. There are five types of blocks corresponding to the five business logics in the circuit as previously mentioned. A certain type of work is packaged into one block, which improves the efficiency a lot by batch verify.

- `generateProof()`

  The operator is responsible for maintaining the state of the entire exchange off the chain. The Merkle tree is updated by user actions such as deposit, withdrawal, order settlement or cancellation. The correctness of state update is checked in circuit code, which is also called constraints in a circuit. Thus the operator must update the states according to the actual operation of users. The zkSNARKs circuit code is implemented with the `EthSnarks` library. Each circuit requires a `trusted setup` to generate a pair of `proving key` and `verification key`. The operator needs to generate proof each time with the corresponding `providing key`.

- `verifyBlcoks()`

  The operator generates a zkSNARKs proof for each block off the chain and submits it to the contract. The proof is verified with the corresponding `verification key`. It takes a relatively long time to generate a proof. Thus submission and verification of a block are separated here. The operator could also submit proofs in disorder. A block could have three status as `COMMITTED`, `VERIFIED`, and

`FINALIZED` and could be changed from `COMMITTED` to `VERIFIED` when the proof is accepted by contract. If all blocks before some block have been verified, they are all considered to be `FINALIZED`. If the exchange is created with data-availability enabled, the operator must also submit more detailed data when `commitBlock()` to get the block verified successfully later.

- `deposit()`

  The user deposits to exchange by calling the contract. The operator must process the request within a certain period and package it into the block for submitting and verifying.

- `withdraw()`

  Users have several ways to withdraw and the two most commonly used are called `OnchainWithdrawal` and `OffchainWithdrawal`. Users do not have to call contracts in off-chain mode, which is more convenient except there is a potential risk that the operator could deliberately not process the request. Users could manually make a withdrawal request in the contract. This on-chain request will be processed within a certain period (just like deposit function).

  Particularly, Loopring V3 also considers the safety of user assets in extreme cases such as the operator stopping the maintenance of the exchange contract and also abandoning its staking and the exchange entering into the withdrawal mode. Users only need to provide the Merkle proof of their assets to withdraw their assets properly.

4. Audit Detail

This part describes the process and detailed results of the audit, also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Labs. We analyzed the project from code bug, logical implementation and potential risks. The process consists of four steps:

- Fully analysis of code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the source code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line and the result could be categorized into twenty-one types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g. overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓

18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Definitions and implementations of `getLRCFeeStats()` interface are inconsistent

Risk Type	Risk Level	Impact	Status
Implementation	Info	Inconsistency of document and implementation	Updated

Description

The documentation of `getLRCFeeStats()` in `IProtocolFeeVault` does not match with the implementation, which could be in the wrong sequence.

```
    /// @return accumulatedDAOFund The accumulated amount of LRC to
burn.
    /// @return accumulatedBurn The accumulated amount of LRC as
developer pool.
```

Suggestion

Fix the comments.

Status

It has been corrected in the new version by developers themselves.

4.3.2 Assembly code in `SignatureBasedAddressWhitelist` does not work

Risk Type	Risk Level	Impact	Status
Implementation	Low	Functional fail	Updated

Description

```
    assembly {
        t := mload(add(permission, 8)) // first 8 bytes as time in
second since epoch
        ...
    }
```

Comments said to take `first 8 bytes`. But the actual value contains the length information of the `permission` variable. So the `t` variable is completely wrong. Besides, this single contract lacks test cases.

Suggestion

The extracted data should be truncated. Write test cases for this single contract.

Status

It has been corrected in the new version by developers themselves.

4.3.3 Operator influence on the choice of maker/taker orders

Risk Type	Risk Level	Impact	Status
Mechanism Design	Low	Affect the transaction price within a limited range	Discussed

Description

The design doc writes about the choice of maker/taker orders as follows.

> *The operator chooses which order is the maker and which order is the taker. The first order in the ring is always the taker order, the second order is always the maker order. We always use the rate of the second order for the trade.*

Is there any room for the operator to manipulate?

A ring matching example. Both two orders are `buy` orders. The `buy order` option is designed to help takers earn spreading.

Case A	Taker (A)	Maker (B)
tokenS	WETH	GTO
tokenB	GTO	WETH

amountS	101	200
amountB	100	200
isBuy	true	true
fill.S	101 -> 100 (spread = 1)	100
fill.B	100	100
target price (WETH/GTO)	1.01	1
settle price (WETH/GTO)	1	1
balance tokenS	1	100
balance tokenB	100	100

In this case, the operator settles the ring at the price `1` (WETH/GTO). The taker (A) got the spreading.

If the operator just switches the role of taker and maker.

Case B	**Taker (B)**	**Maker (A)**
tokenS	GTO	WETH
tokenB	WETH	GTO
amountS	200	101
amountB	200	100
isBuy	true	true
fill.S	101 -> 100 (spread = 1)	101
fill.B	101	100
target price (WETH/GTO)	1	1.01
settle price (WETH/GTO)	1.01	1.01

balance tokenS	100	0
balance tokenB	101	100

In this case, the operator settles the ring at the price `1.01` (WETH/GTO). The taker (B) got the spreading.

So, the operator could choose which user to get a better price.

Status

Brecht Devos replied in the issue#623。

The operator has the right to decide on order-matching by design. It is acceptable that an operator could affect the price in a limited range since no user got a price worse than expected.

4.3.4 Unchecked return values in use of `transferTokens()`

Risk Type	Risk Level	Impact	Status
Best Practice	Info	Avoid the best practice	To Be Discussed

Description

Return values are unchecked in `withdrawFromMerkleTreeFor()` and `withdrawFromDepositRequest()` function.

```
    // Transfer the tokens
    transferTokens(
        S,
        _deposit.accountID,
        _deposit.tokenID,
        amount,
        false
    );
```

The condition `allowFailure = false` is used in these two cases and the transaction will be reverted in advance when something wrong in `transferTokens()` function. So there is no risk in not checking return values here.

Suggestion

Add a check or comment description.

Status

To be discussed.

4.3.5 Unclear intention of `transferDeposit()` function

Risk Type	Risk Level	Impact	Status
Implementation	Medium	Ambiguity	Updated

Description

We take Ethers from `msg.sender` but tokens from `accountOwner` in `transferDeposit()` function.

```
    function transferDeposit(
        address accountOwner,
        ...
        )
        private
    {
        ...
        require(msg.value >= totalRequiredETH, "INSUFFICIENT_FEE");
        uint feeSurplus = msg.value.sub(totalRequiredETH);
        msg.sender.transferETH(feeSurplus, gasleft());
        ...
                tokenAddress.safeTransferFrom(
                    accountOwner,
                    address(this),
                    amount
                )
        ...
    }
```

In the scenario of deposit, it seems to be unreasonable to take assets from two accounts. The current implementation is that someone else pays the ETH fee for `accountOwner` and updates the deposit with assets of `accountOwner` himself. The original intent could be someone depositing for others with his balance.

Suggestion

Remove the meaningless `accountOwner` parameter directly in `transferDeposit()` function and use `msg.sender` as first argument in `safeTransferFrom()`.

Status

It has been updated in the new version by developers themselves. The `msg.sender` is used in `transferDeposit()` as first argument. We recommend to remove it directly.

4.3.6 Consider about overflow issues of the timestamp in `Block`

Risk Type	Risk Level	Impact	Status
Protocol Design	Low	Time overflow, affecting protocol stability	To be discussed

Description

The timestamp is stored in `uint32` in `Block` data structure and there is a risk of overflow. The time variable could not be represented correctly after the year 2106.

```
struct Block
{
    // The time the block was created.
    uint32 timestamp;
}
```

Bitcoin also uses the `uint32` type for timestamp while Ethereum uses `uint256` instead. Nonetheless, there should be enough time for the protocol to upgrade.

Suggestion

Know about this issue and discuss the possibility and necessity of replacing `uint32` with `uint256`.

Status

To be discussed.

4.3.7 `TransformRingSettlementDataGadget` could be optimized

Risk Type	Risk Level	Impact	Status
Circuit Code	Info	Redundant Code	To be discussed

Description

In `RingSettlementCircuit.h`, the `generate_r1cs_constraints()` function of `TransformRingSettlementDataGadget` could be further optimized into the following code.

```
struct Range
{
    unsigned int offset;
    unsigned int length;
};

std::vector<Range> ranges;
ranges.push_back({{0, 40}});                    // orderA.orderID +
orderB.orderID
ranges.push_back({{40, 40}});                   // orderA.accountID +
orderB.accountID
ranges.push_back({{80, 8}, {120, 8}});          // orderA.tokenS +
orderB.tokenS
ranges.push_back({{88, 24},{128, 24}});         // orderA.fillS +
orderB.fillS
ranges.push_back({{112, 8}});                   // orderA.data
ranges.push_back({{152, 8}});                   // orderB.data
for(const Range& subRange : ranges)
{

    for (unsigned int i = 0; i < numRings; i++)
    {
        transformedData.add(subArray(data, i * ringSize +
subRange.offset, subRange.length));
    }
}
```

There is no need to do data compression work in the circuit. The `compressedData` variable and related code are useless. The original code introduces more redundant operations on data.

Suggestion

Consider removing the `compressedData` related code.

Status

To be discussed.

4.3.8 Wrong comment in `TakerMakerMatchingGadget`

Risk Type	Risk Level	Impact	Status
Documentation	Info	Wrong comments of code	To be discussed

Description

```
takerFillB_lt_makerFillS(pb, takerFill.B, makerFill.S,
NUM_BITS_AMOUNT, FMT(prefix, ".takerFill.B < makerFill.B")),
```

Suggestion

Correct to `.takerFill.B < makerFill.S`.

Status

To be discussed.

4.3.9 Call logic of `generateKeyPair()` is impractical

Risk Type	Risk Level	Impact	Status
Circuit Code	Low	Wrong comments of code	Updated

Description

The call logic of `generateKeyPair()` in `main.cpp` of circuit code does not meet requirements of production.

```
    if (mode == Mode::CreateKeys || mode == Mode::Prove)
    {
        if (!generateKeyPair(pb, baseFilename))
        {
            std::cerr << "Failed to generate keys!" << std::endl;
            return 1;
        }
    }
```

Whenever the `Mode` is `Prove` or `CreateKeys`, the circuit code always tries to `generateKeyPair()`. A trusted setup is required in the actual scenario of production for `generateKeyPair()`, which needs more people to join. The logic of regenerate `KeyPair` when it is not found is not realistic.

Suggestion

Separate `CreateKeys` and `Prove` actions. When `pk` and `vk` do not exist, the `Prove` operation should be not allowed.

Status

It has been updated in the new version by developers themselves.

4.3.10 Redundant fields in the `OffchainWithdrawalBlock` class

Risk Type	Risk Level	Impact	Status
Circuit Code	Low	Redundant Fields	To Be Discussed

Description

The `startIndex` and `count` fields are not used in `OffchainWithdrawalBlock` class of `Data.h` file.

```
class OffchainWithdrawalBlock
{
public:
    ethsnarks::FieldT exchangeID;
    ethsnarks::FieldT merkleRootBefore;
    ethsnarks::FieldT merkleRootAfter;
    libff::bigint<libff::alt_bn128_r_limbs> startHash;
    ethsnarks::FieldT startIndex; // unused
    ethsnarks::FieldT count; // unused
    ethsnarks::FieldT operatorAccountID;
    AccountUpdate accountUpdate_O;
    std::vector<Loopring::OffchainWithdrawal> withdrawals;
};
```

Suggestion

Remove unused fields.

Status

To be discussed.

4.4 Risks

4.4.1 Consider about exception handing in extreme cases

Risk Type	Risk Level	Impact	Status
Parameter Settings	Medium	Protocol Stability	To Be Discussed

Description

Currently a lot of time limit related parameters in `ExchangeData.sol` are hardcoded, which are unchangeable after contract deployment. The exchange contract has strict time limits on the processing of on-chain requests and the time validity of orders. It is necessary to consider whether the protocol can still operate stably under extreme circumstances. Most of these parameters also involve a power restriction of operators, which should be chosen very carefully.

Key parameters such as `MAX_PROOF_GENERATION_TIME_IN_SECONDS`, `MAX_AGE_REQUEST_UNTIL_WITHDRAW_MODE`, `TIMESTAMP_HALF_WINDOW_SIZE_IN_SECONDS` need special attention.

Suggestion

- Think about and simulate extreme conditions
- Choose values based on special scenarios such as an excessive volume of transactions, crowded Ethereum network, unstable backend for operators, etc.
- The backend needs to be tested thoughtfully and to be robust enough to deal with instability or congestion of blockchain network, transaction sending, and monitoring, chain fork processing, anti-DOS attacks, etc.

Status

To be discussed.

5. Conclusion

The Loopring Protocol (V3) takes advantage of zkSNARKs technology to completely design and implement a high-performance decentralized exchange protocol. After auditing and analyzing the contract and circuit code of it, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. Particularly, SECBIT Labs holds the view that the Loopring Protocol (V3) project has a high quality of code with a clean structure, good function naming conventions, complete annotations, and well-designed test cases. For the first time, Loopring implements the circuit code of complicated on-chain businesses for real, which resulting in significant compression of on-chain computations. They have achieved great improvements in the practical use of zero-knowledge proof techniques. It's impressive to see the pursuit of both performance and decentralization together with the efforts to secure user assets, which are all happening to this protocol. The experience of their research on layer-2 for scaling is valuable to the whole community.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company or investment.

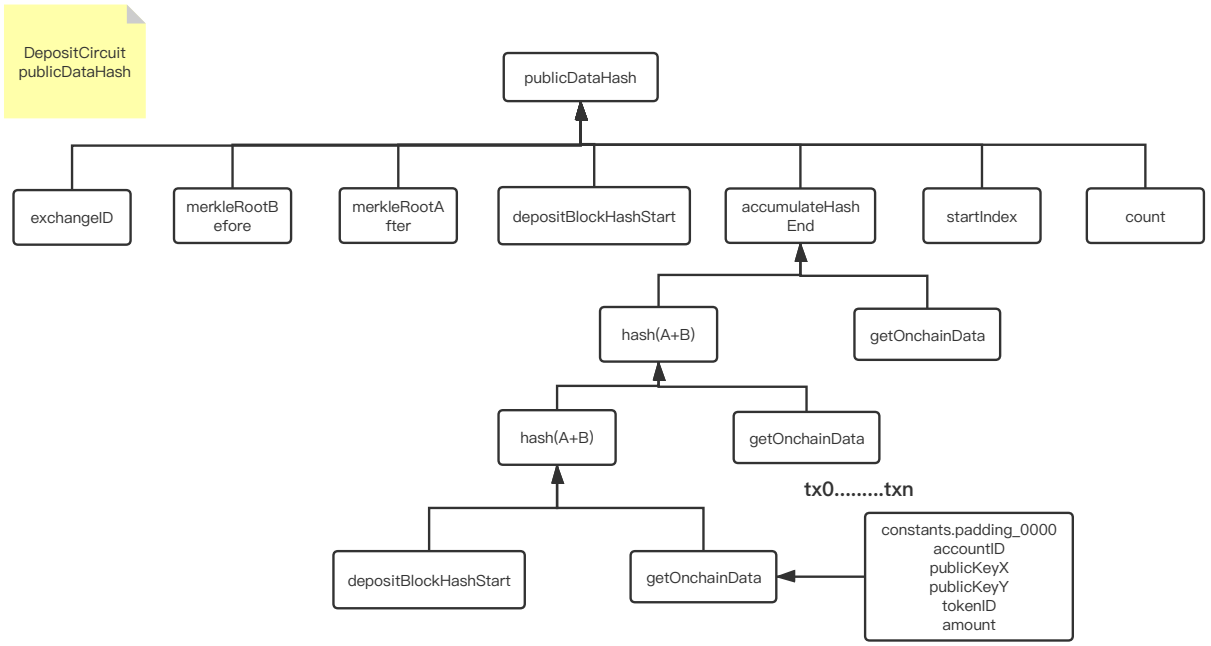
APPENDIX

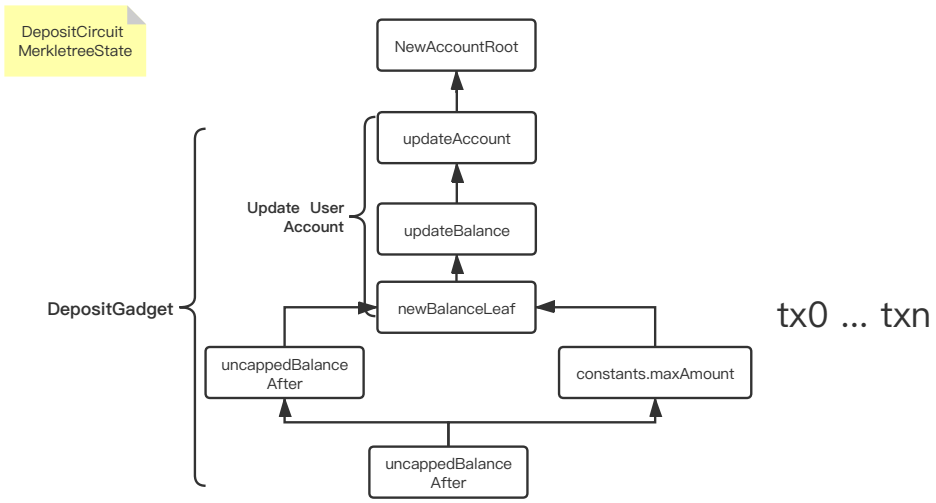
Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality could possibly bring risks.

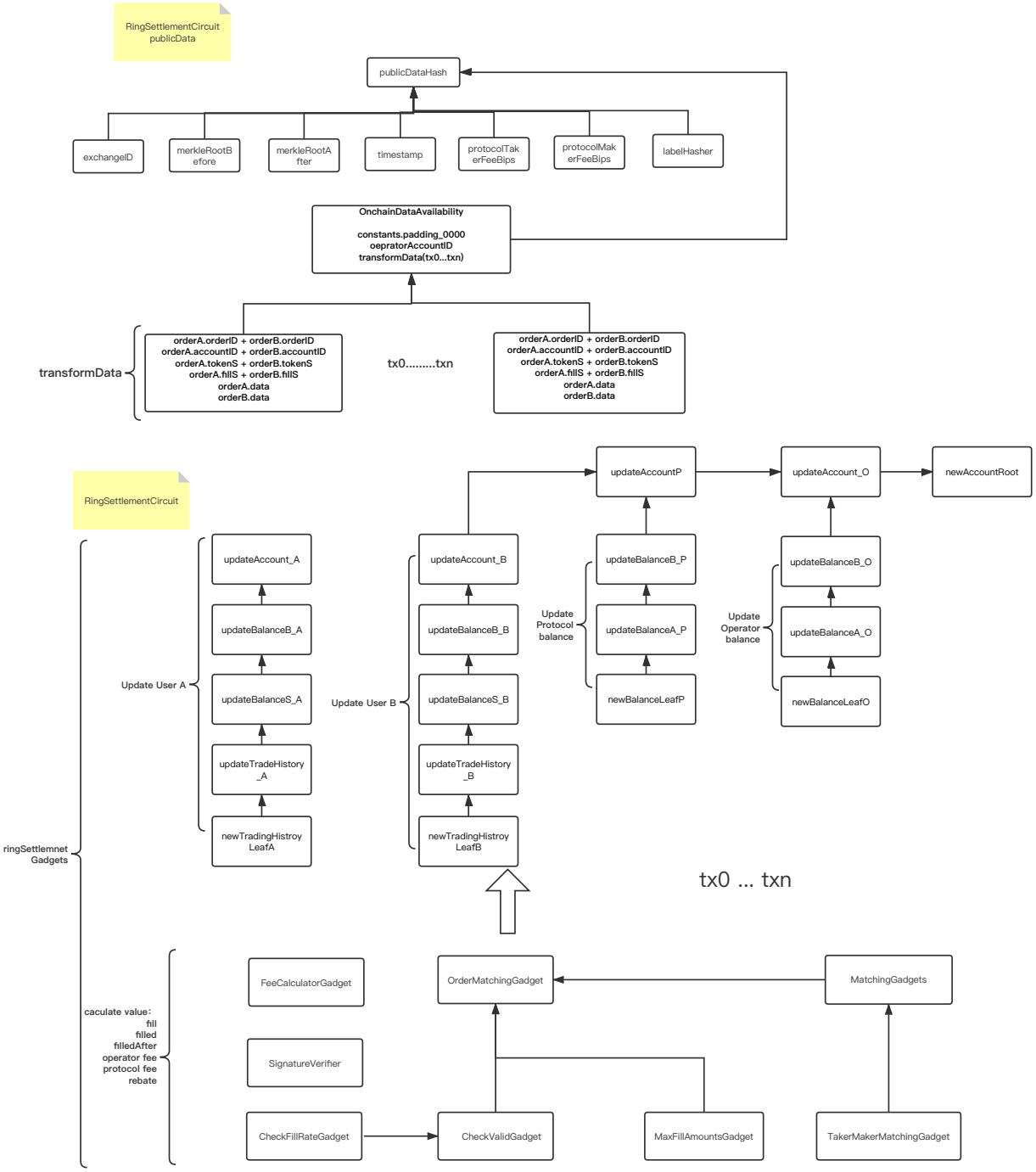
Analysis Diagram on Circuit Code

DepositCircuit

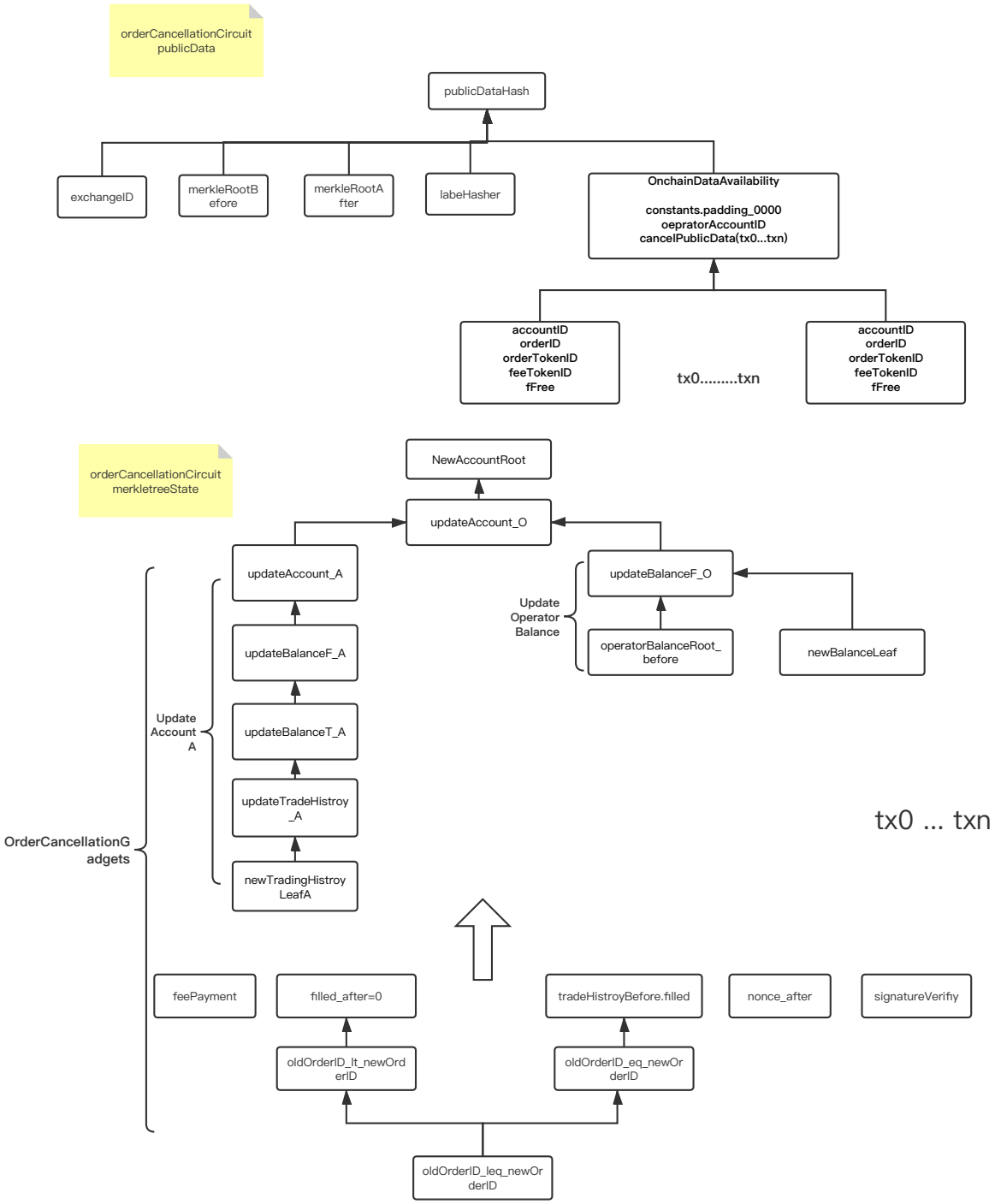




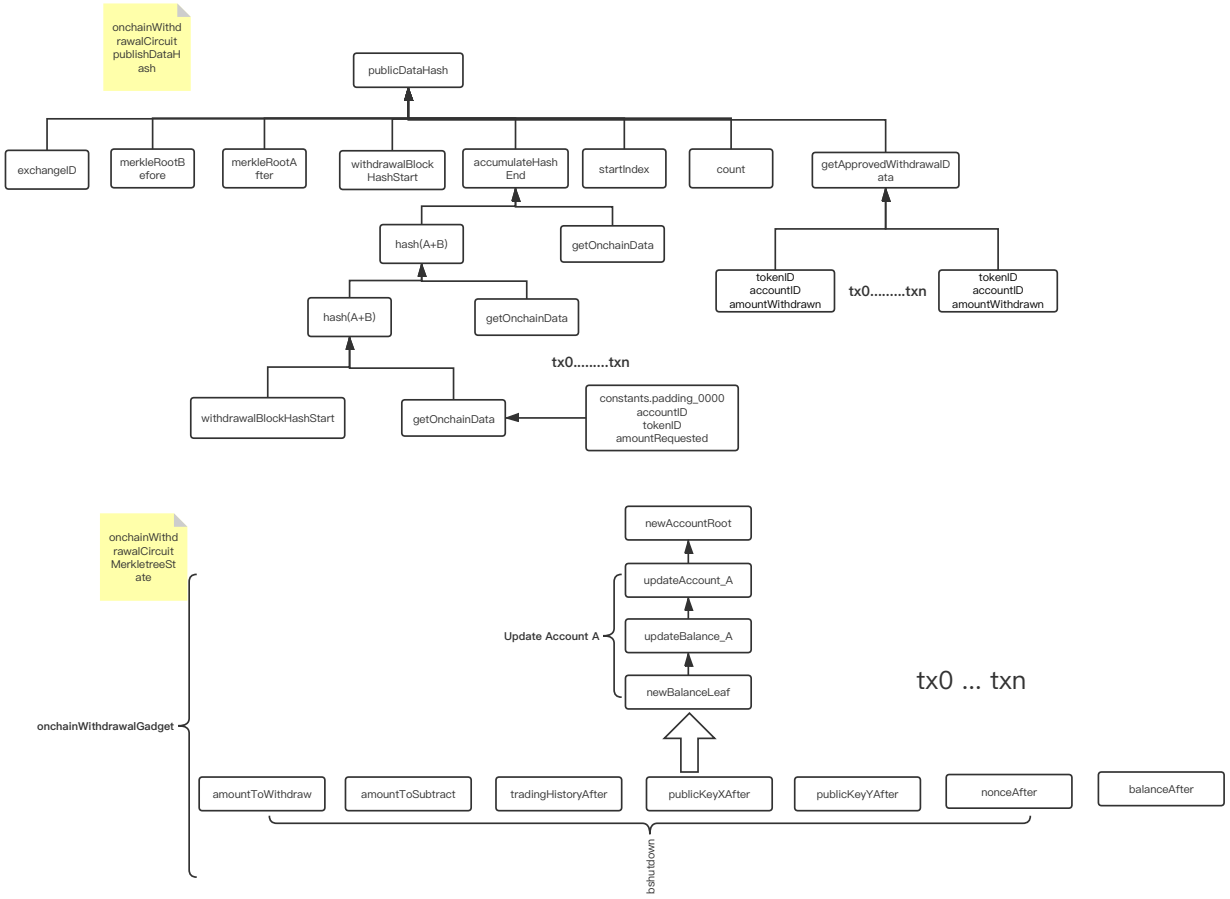
RingsettlementCircuit



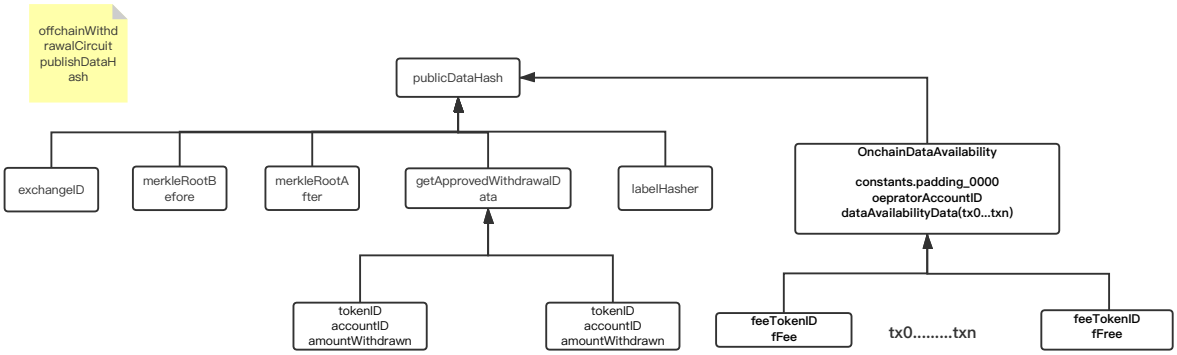
OrderCancellationCircuit

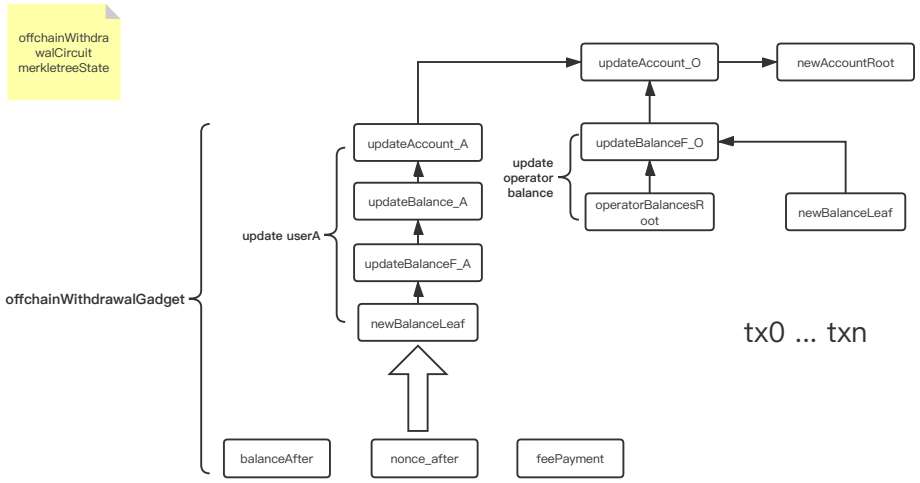


OnchainWithdrawalCircuit



OffchainWithdrawalCircuit





SECBIT Labs is devoted to constructing a common-consensus, reliable and ordered blockchain economic entity.

 http://www.secbit.io

 audit@secbit.io

 @secbit_io