Swop Fi Audit Report

Version 1.0.2

Serial No. 2021071100022018

Presented by Fairyproof

July 11, 2021

FAIRYPROOF

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the SwopFi project, at the request of the SwopFi team.

Project Token's Name:

SWOP

Audited Code's Github Repository:

https://github.com/swopfi/swopfi-smart-contracts/tree/master/dApps

The audited files include all the files with the extension of ".ride" under the dApps directory and its sub-directories.

Audited Code's Github Commit Number:

9fbbc63c75674d902d604b979570d5415d9bbd6e

Audited Source Files' Onchain Address:

N/A

Audited Source Files:

The source files audited include all the files with the extension of ".ride" as follows:

```
dApps/
├── SWOP
|   ├── earlybirds.ride
|   ├── farming.ride
|   ├── governance.ride
|   ├── oracle.ride
|   ├── voting.ride
|   ├── voting_for_new_pool.ride
|   └── wallet_verifier.ride
├── flat.ride
├── nsbt_usdn.ride
├── other_cpmm.ride
└── waves_eurn.ride
```

The goal of this audit is to review SwopFi's Ride implementation for its DEX application, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Fairyproof to make sure we understand the size, scope, and functionality of the project's source code.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to

identify potential vulnerabilities.
iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Fairyproof describe.

2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run the test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the source code to improve maintainability, security, and control based on the established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following sources of truth about how the SwopFi system should work:

https://swop.fi/

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the SwopFi team or reported an issue.

— Comments from Auditor

No vulnerabilities with critical, high, medium or low-severity were found in the above source code.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to SwopFi

Swop.fi is a service featuring functionality for instant exchange of cryptocurrencies and investing assets in order to receive passive income. The service is based on the Waves blockchain which provides high transaction speed and low network fees.

04. Major functions of audited code

The audited code implements the following functions:

- token issurance. Issurance of the governance token : SWOP, Max supply: 6 million
- early-bird program. This program is to incentivize users to provide liquidity to code-start the exchange.
- a DEX based on CPMM and Flat with low slippages
- liquidity mining
- governance.

05. Key points in audit

During the audit we executed the following work items:

- automated code review
- manual code review
- logic analysis
- unit testing
- system testing

06. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Re-entrancy Attack
- DDos Attack
- Integer Overflow
- Function Visibility
- Logic Vulnerability
- Uninitialized Storage Pointer
- Arithmetic Precision
- Tx.origin
- Shadow Variable
- Design Vulnerability
- Token Issurance
- Asset Security
- Access Control

07. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

08. Major areas that need attention

Based on the provided souce code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Token Issurance

We checked whether or not the contract files could mint tokens at will.

We didn't find issues or risks in these functions or areas at the time of writing.

- Crypto Exchange

We checked whether or not there were potential issues or risks in the CPMM and Flat based crypto exchange.

We didn't find issues or risks in these functions or areas at the time of writing.

- Liquidity Mining

We checked whether or not users' assets in the mining pools were safe.

We didn't find issues or risks in these functions or areas at the time of writing.

- Early-bird Program

We checked whether or not the code implementation in the early-bird program worked as the design logic.

We didn't find issues or risks in these functions or areas at the time of writing.

- Governance

We checked whether or not there were potential issues or risks in the governance module.

We didn't find issues or risks in these functions or areas at the time of writing.

- Miscellaneous

We didn't find issues or risks in other functions or areas at the time of writing.

09. Key Audit Details and Analysis

- Flat

Fairyproof's Analysis:

The following code is to implement a flat mechanism. That is when exchanging small amounts, the price should be as close as possible to the constant price described by the formula designed by the Swop team:

The function used here is : (x + y) * s^(- α) + 2 * (xy)^(1/2) * (s - β)^α = k,

where x and y are the two tokens in a trading pair and s=(x/y+y/x)/2

when α and β are 0.5 and 0.46 respectively, they result in the least slippage at a balance ratio close to equilibrium.

Audit Result: No Issue

Code section is as follows:

{-# STDLIB_VERSION 4 #-}

{-# CONTENT_TYPE DAPP #-}

{-# SCRIPT_TYPE ACCOUNT #-}

let version = "2.0.0"

let kVersion = "version"

let kActive = "active"

let kAssetIdA = "A_asset_id"

let kAssetIdB = "B_asset_id"

let kBalanceA = "A_asset_balance"

let kBalanceB = "B_asset_balance"

let kShareAssetId = "share_asset_id"

let kShareAssetSupply = "share_asset_supply"

let kFee = "commission"

let kFeeScaleDelimiter = "commission_scale_delimiter"

```
let kInvariant = "invariant"
let kCause = "shutdown_cause"

let adm1 = base58'DXDY2itiEcYBtGkVLnkpHtDFyWQUkoLJz79uJ7ECbMrA'
let adm2 = base58'E6Wa1SGoktYcjHjsKrvjMiqJY3SWmGKcD8Q5L8kxSPS7'
let adm3 = base58'AZmWJtuy4GeVrMmJH4hfFBRApe1StvhJSk4jcbT6bArQ'
let admStartStop = base58'EtVkT6ed8GtbUiVVEqdmEqsp2J4qbb3rre2HFgxeVYdg'
let admStaking = base58'Czn4yoAuUZCVCLJDRfskn8URfkwpknwBTZDbs1wFrY7h'

let govAddr = Address(base58'3P6J84oH51DzY6xk2mT5TheXRbrCwBMxonp')
let stakingAddress = Address(base58'3PNikM6yp4NqcSU8guxQtmR5onr2D4e8yTJ')
```

- Checkpoint1: Set Sponsor Fee

```
let USDN = base58'DG2xFkPdDwKUoBkzGAhQtLpSGzfXLiCYPEzeKH2Ad24p'
let stakingFeeInUSDN = 9 * assetInfo(USDN).value().minSponsoredFee.value()

let isActive = this.getBooleanValue(kActive)

let strAssetIdA = this.getStringValue(kAssetIdA)
let strAssetIdB = this.getStringValue(kAssetIdB)

let assetIdA = if strAssetIdA == "WAVES" then unit else strAssetIdA.fromBase58String()
let assetIdB = if strAssetIdB == "WAVES" then unit else strAssetIdB.fromBase58String()
```

- Checkpoint2: Get Token's Name

```
let assetNameA = match assetIdA {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
let assetNameB = match assetIdB {
```

case id: ByteVector => assetInfo(id).value().name

case waves: Unit => "WAVES"

}

- Checkpoint3: Get Token Balance

let balanceA = this.getIntegerValue(kBalanceA)

let balanceB = this.getIntegerValue(kBalanceB)

let shareAssetId = this.getStringValue(kShareAssetId).fromBase58String()

let shareAssetSupply = this.getIntegerValue(kShareAssetSupply)

let invariant = this.getIntegerValue(kInvariant)

let fee = 500 # fee/feeScale6 = 0.0005

let feeGovernance = 200 # feeGovernance/feeScale6 = 0.0002. 3/5 fee back to the dApp, 2/5 sends to the governance address

let feeScale6 = 1000000

let scale3 = 1000

let scale8 = 100000000

let scale12 = 1000000000000

let slippageScale3 = 1000

let digits8 = 8

let dAppThreshold = 50 # dAppThresholdAmount/dAppThresholdAmountDelimiter = 0.5

let dAppThresholdScale2 = 100

let exchangeRatioLimitMin = 90000000 # 0.9*scale8. This parameter helps to avoid losses when an incorrect argument is passed

let exchangeRatioLimitMax = 110000000 #1.1*scale8. This parameter helps to avoid losses when an incorrect argument is passed

let alpha = 50 # model coefficient alpha = 0.15 with 2 digits

let alphaDigits = 2

let beta = 46000000 # model coefficient beta = 0.46 with 8 digits

func accountBalance(assetId: ByteVector|Unit) = match assetId {

case id: ByteVector => this.assetBalance(id)

case waves: Unit => this.wavesBalance().available

}

- Checkpoint4: Return Staked USDN

let stakedAmountUSDN = match stakingAddress.getInteger("rpd_balance" + *USDN.toBase58String()* + "" + this.toString()) {

case staked: Int => staked

case nothing: Unit => 0

}

- Checkpoint5: Get USDN Balance

let availableBalanceA = balanceA - if assetIdA == USDN then stakedAmountUSDN else 0

let availableBalanceB = balanceB - if assetIdB == USDN then stakedAmountUSDN else 0

- Checkpoin6: Get Staked Balance

let accountBalanceWithStakedA = accountBalance(assetIdA) + if assetIdA == USDN then stakedAmountUSDN else 0

let accountBalanceWithStakedB = accountBalance(assetIdB) + if assetIdB == USDN then stakedAmountUSDN else 0

let hasEnoughBalance = accountBalanceWithStakedA >= balanceA && accountBalanceWithStakedB >= balanceB

skewness = 0.5*(x/y+y/x)

Each fraction multiple by scale in order to avoid zeroing

During working on this fraction scale8 was switched to scale12. To unscale back to 8 digits added /10000

```
func skewness(x: Int, y: Int) = (scale12.fraction(x, y) + scale12.fraction(y, x)) / 2 / 10000

# Calculate (x+y)skewness^(-alpha) + 2(xy)^0.5(skewness-beta)^alpha
func invariantCalc(x: Int, y: Int) = {
let sk = skewness(x, y)
fraction(
x + y,
scale8,
pow(sk, digits8, alpha, alphaDigits, digits8, UP)
) + 2 * fraction(
pow(fraction(x, y, scale8), 0, 5, 1, digits8 / 2, DOWN),
pow(sk - beta, digits8, alpha, alphaDigits, digits8, DOWN),
scale8
)
}

func calculateSendAmount(amountToSendEstimated: Int, minTokenReceiveAmount: Int,
tokenReceiveAmount: Int, tokenId: ByteVector|Unit) = {
let slippageValue = scale8 - scale8 * 1 / 10000000 # 0.000001% of slippage
let deltaBetweenMaxAndMinSendValue = amountToSendEstimated - minTokenReceiveAmount

# only one of the variables will be used depending on the token
let x = balanceA + tokenReceiveAmount
let y = balanceB + tokenReceiveAmount

let invariantNew =
if tokenId == assetIdA then
invariantCalc(x, balanceB - amountToSendEstimated)
else if tokenId == assetIdB then
invariantCalc(balanceA - amountToSendEstimated, y)
else throw("Wrong asset in payment")
```

```
let invariantEstimatedRatio = scale8.fraction(invariant, invariantNew)

func getStepAmount(acc: Int, step: Int) = {
if acc == -1 then
let amountToSend = amountToSendEstimated - step * deltaBetweenMaxAndMinSendValue / 5
let stepInvariant =
if tokenId == assetIdA then
invariantCalc(x, balanceB - amountToSend)
else invariantCalc(balanceA - amountToSend, y)

if stepInvariant > invariant then
amountToSend
else
-1
else
acc
}

let stepAmount = FOLD<5>([1, 2, 3, 4, 5], -1, getStepAmount)
if stepAmount < 0 then
throw("something went wrong while working with amountToSend") # TODO when?
else if invariantEstimatedRatio > slippageValue && invariantNew > invariant then
amountToSendEstimated
else
stepAmount
}

func getAssetInfo(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector =>
let stringId = id.toBase58String()
```

```
let info = assetInfo(id).valueOrErrorMessage("Asset " + stringId + " doesn't exist")
(stringId, info.name, info.decimals)
case waves: Unit => ("WAVES", "WAVES", 8)
}

func suspend(cause: String) = [
BooleanEntry(kActive, false),
StringEntry(kCause, cause)
]

func deductStakingFee(amount: Int, assetId: ByteVector|Unit) =
if assetId == USDN then {
let result = amount - stakingFeeInUSDN

if result <= 0 then
throw("Insufficient amount " + amount.toString()
+ " to deduct staking fee " + stakingFeeInUSDN.toString() + " USD-N")
else result
} else amount

func throwIsActive() = throw("DApp is already active")
func throwIsInactive() = throw("DApp is inactive at this moment")
func throwOnlyAdmin() = throw("Only admin can call this function")
func throwAssets() = throw("Incorrect assets attached. Expected: " + strAssetIdA + " and " + strAssetIdB)
func throwThreshold(threshold: Int, amountA: Int, amountB: Int) = throw("New balance in assets of the DApp is less than threshold " + threshold.toString()
+ ": " + amountA.toString() + " " + assetNameA + ", " + amountB.toString() + " " + assetNameB)
func throwInsufficientAvailableBalance(amount: Int, available: Int, assetName: String) = throw("Insufficient DApp balance to pay "
+ amount.toString() + " " + assetName + " due to staking. Available: "
+ available.toString() + " " + assetName + ". Please contact support in Telegram: https://t.me/swopfisupport")
```

```
func throwInsufficientAvailableBalances(amountA: Int, amountB: Int) = throw("Insufficient DApp balance to pay "

+ amountA.toString() + " " + assetNameA + " and " + amountB.toString() + " " + assetNameB

+ " due to staking. Available: "

+ availableBalanceA.toString() + " " + assetNameA + " and " + availableBalanceB.toString() + " " + assetNameB

+ ". Please contact support in Telegram: https://t.me/swopfisupport")

func suspendSuspicious() = suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " + balanceB.toString() + " " + assetNameB

+ ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA + ", " + accountBalance(assetIdB).toString() + " " + assetNameB)

@Callable(i)

func init() = {

let (pmtAmountA, pmtAssetIdA) = (i.payments[0].amount, i.payments[0].assetId)

let (pmtAmountB, pmtAssetIdB) = (i.payments[1].amount, i.payments[1].assetId)

let (pmtStrAssetIdA, pmtAssetNameA, pmtDecimalsA) = getAssetInfo(pmtAssetIdA)

let (pmtStrAssetIdB, pmtAssetNameB, pmtDecimalsB) = getAssetInfo(pmtAssetIdB)


if this.getBoolean(kActive).isDefined() then

throwIsActive()

else if pmtAssetIdA == pmtAssetIdB then

throw("Assets must be different")

else {

let shareName = "s" + pmtAssetNameA.take(7) + "_" + pmtAssetNameB.take(7)

let shareDescription = "ShareToken of SwopFi protocol for " + pmtAssetNameA + " and " + pmtAssetNameB + " at address " + this.toString()


# we save 16 digit info with using digits8 in pow functions. We divide /scale8 to delete 8 digits and another 8 digits we define as digits in shareToken in Issue tx

let shareDecimals = (pmtDecimalsA + pmtDecimalsB) / 2

let shareInitialSupply = fraction(

pow(pmtAmountA, pmtDecimalsA, 5, 1, pmtDecimalsA, HALFDOWN),

pow(pmtAmountB, pmtDecimalsB, 5, 1, pmtDecimalsB, HALFDOWN),
```

```
pow(10, 0, shareDecimals, 0, 0, HALFDOWN)
)
let shareIssue = Issue(shareName, shareDescription, shareInitialSupply, shareDecimals, true)
let shareIssueId = shareIssue.calculateAssetId()

let invariantCalculated = invariantCalc(pmtAmountA, pmtAmountB)
[
StringEntry(kVersion, version),
BooleanEntry(kActive, true),
StringEntry(kAssetIdA, pmtStrAssetIdA),
StringEntry(kAssetIdB, pmtStrAssetIdB),
IntegerEntry(kBalanceA, pmtAmountA),
IntegerEntry(kBalanceB, pmtAmountB),
IntegerEntry(kInvariant, invariantCalculated),
IntegerEntry(kFee, fee),
IntegerEntry(kFeeScaleDelimiter, feeScale6),
shareIssue,
StringEntry(kShareAssetId, shareIssueId.toBase58String()),
IntegerEntry(kShareAssetSupply, shareInitialSupply),
ScriptTransfer(i.caller, shareInitialSupply, shareIssueId)
]
}
}


# if someone ready to replenish in both tokens. It's necessary to bring tokens amount in proportion according to dApp state
@Callable(i)
func replenishWithTwoTokens(slippageTolerance: Int) = {
let pmtAssetIdA = i.payments[0].assetId
let pmtAssetIdB = i.payments[1].assetId
# block for accounting the cost of fees for staking operations
```

```
let pmtAmountA = deductStakingFee(i.payments[0].amount, pmtAssetIdA)

let pmtAmountB = deductStakingFee(i.payments[1].amount, pmtAssetIdB)


# fraction should be equal 1(multiple by 1000) if depositor replenish with proportion according to actual price

let tokenRatio = scale8.fraction(balanceA, pmtAmountA)

.fraction(scale3, scale8.fraction(balanceB, pmtAmountB))


let ratioShareTokensInA = scale8.fraction(pmtAmountA, balanceA)

let ratioShareTokensInB = scale8.fraction(pmtAmountB, balanceB)

let shareTokenToPayAmount = min([ratioShareTokensInA, ratioShareTokensInB]).fraction(shareAssetSupply, scale8)


let invariantCalculated = invariantCalc(balanceA + pmtAmountA, balanceB + pmtAmountB)


if !isActive then

throwIsInactive()

else if slippageTolerance < 0 || slippageTolerance > 10 then

throw("Slippage tolerance must be <= 1%")

else if i.payments.size() != 2 then

throw("Two attached assets expected")

else if pmtAssetIdA != assetIdA || pmtAssetIdB != assetIdB then

throwAssets()

else if tokenRatio < (scale3 * (slippageScale3 - slippageTolerance)) / slippageScale3

|| tokenRatio > (scale3 * (slippageScale3 + slippageTolerance)) / slippageScale3 then

throw("Incorrect assets amount: amounts must have the contract ratio")

else if shareTokenToPayAmount == 0 then

throw("Too small amount to replenish")

else if !hasEnoughBalance then

suspendSuspicious()

else [

IntegerEntry(kBalanceA, balanceA + pmtAmountA),
```

```
IntegerEntry(kBalanceB, balanceB + pmtAmountB),
IntegerEntry(kShareAssetSupply, shareAssetSupply + shareTokenToPayAmount),
IntegerEntry(kInvariant, invariantCalculated),
Reissue(shareAssetId, shareTokenToPayAmount, true),
ScriptTransfer(i.caller, shareTokenToPayAmount, shareAssetId)
]
}

# if someone ready to replenish in one token
@Callable(i)
func replenishWithOneToken(virtualSwapTokenPay: Int, virtualSwapTokenGet: Int) = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)

let pmtMinThreshold = 5000000
let thresholdValueForMinTolerance = 50000000
let tolerance = if pmtAmount < thresholdValueForMinTolerance then 100000 else 1

let slippageValueMinForReplenish = scale8 - scale8 * tolerance / 10000000 # if pmtAmount > 50.000000 then slippage = 0.000001%
let slippageValueMaxForReplenish = scale8 + scale8 * tolerance / 10000000

let slippageValueMinForSwap = scale8 - scale8 * 1 / 10000000

if !isActive then
throwIsInactive()
else if pmtAmount < pmtMinThreshold then
throw("Payment amount " + pmtAmount.toString() + " does not exceed the minimum amount of " + pmtMinThreshold.toString() + " tokens")
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if !hasEnoughBalance then
suspendSuspicious()
```

```
else if pmtAssetId != assetIdA && pmtAssetId != assetIdB then
throwAssets()
else {
let (virtualReplenishA, virtualReplenishB,
balanceAfterSwapA, balanceAfterSwapB,
invariantCalculated,
newBalanceA, newBalanceB) =
if pmtAssetId == assetIdA then (
pmtAmount - virtualSwapTokenPay, virtualSwapTokenGet,
balanceA + virtualSwapTokenPay, balanceB - virtualSwapTokenGet,
invariantCalc(balanceA + pmtAmount, balanceB),
balanceA + pmtAmount, balanceB
) else (
virtualSwapTokenGet, pmtAmount - virtualSwapTokenPay,
balanceA - virtualSwapTokenGet, balanceB + virtualSwapTokenPay,
invariantCalc(balanceA, balanceB + pmtAmount),
balanceA, balanceB + pmtAmount
)
let newBalanceEntry =
if pmtAssetId == assetIdA then
IntegerEntry(kBalanceA, newBalanceA)
else IntegerEntry(kBalanceB, newBalanceB)

let invariantNew = invariantCalc(balanceAfterSwapA, balanceAfterSwapB)
let invariantEstimatedRatio = scale8.fraction(invariant, invariantNew)

let ratioVirtualBalanceToVirtualReplenish = (scale8 * scale8).fraction(balanceAfterSwapA,
balanceAfterSwapB)
/ scale8.fraction(virtualReplenishA, virtualReplenishB)
let dAppThresholdAmount = fraction(newBalanceA + newBalanceB, dAppThreshold, 2 *
dAppThresholdScale2)
if invariantEstimatedRatio <= slippageValueMinForSwap || invariantNew < invariant then
```

```
throw("Incorrect virtualSwapTokenPay or virtualSwapTokenGet value")

else if ratioVirtualBalanceToVirtualReplenish < slippageValueMinForReplenish ||
ratioVirtualBalanceToVirtualReplenish > slippageValueMaxForReplenish then

throw("Swap with virtualSwapTokenPay and virtualSwapTokenGet is possible, but ratio after virtual swap is
incorrect")

else if newBalanceA < dAppThresholdAmount || newBalanceB < dAppThresholdAmount then

throwThreshold(dAppThresholdAmount, newBalanceA, newBalanceB)

else {

# fee for staking operation

let ratioShareTokensInA = virtualReplenishA.deductStakingFee(assetIdA).fraction(scale8,
balanceAfterSwapA)

let ratioShareTokensInB = virtualReplenishB.deductStakingFee(assetIdB).fraction(scale8,
balanceAfterSwapB)


let shareTokenToPayAmount = min([ratioShareTokensInA, ratioShareTokensInB]).fraction(shareAssetSupply,
scale8)


[

Reissue(shareAssetId, shareTokenToPayAmount, true),

ScriptTransfer(i.caller, shareTokenToPayAmount, shareAssetId),

IntegerEntry(kShareAssetSupply, shareAssetSupply + shareTokenToPayAmount),

newBalanceEntry,

IntegerEntry(kInvariant, invariantCalculated)

]

}

}

}


@Callable(i)

func withdraw() = {

let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)


# block for accounting the cost of fees for staking operations
```

```
let amountToPayA = pmtAmount.fraction(balanceA, shareAssetSupply).deductStakingFee(assetIdA)
let amountToPayB = pmtAmount.fraction(balanceB, shareAssetSupply).deductStakingFee(assetIdB)

let invariantCalculated = invariantCalc(balanceA - amountToPayA, balanceB - amountToPayB)

if !isActive then
throwIsInactive()
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if pmtAssetId != shareAssetId then
throw("Incorrect asset attached. Expected: " + shareAssetId.toBase58String())
else if !hasEnoughBalance then
suspendSuspicious()
else if amountToPayA > availableBalanceA || amountToPayB > availableBalanceB then
throwInsufficientAvailableBalances(amountToPayA, amountToPayB)
else [
IntegerEntry(kBalanceA, balanceA - amountToPayA),
IntegerEntry(kBalanceB, balanceB - amountToPayB),
IntegerEntry(kShareAssetSupply, shareAssetSupply - pmtAmount),
IntegerEntry(kInvariant, invariantCalculated),
Burn(shareAssetId, pmtAmount),
ScriptTransfer(i.caller, amountToPayA, assetIdA),
ScriptTransfer(i.caller, amountToPayB, assetIdB)
]
}

@Callable(i)
func exchange(estimatedAmountToReceive: Int, minAmountToReceive: Int) = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)
if !isActive then
throwIsInactive()
```

```
else if estimatedAmountToReceive <= 0 then

throw("Estimated amount must be positive. Actual: " + estimatedAmountToReceive.toString())

else if minAmountToReceive > estimatedAmountToReceive then

throw("Minimal amount can't be greater than estimated. Estimated: " +
estimatedAmountToReceive.toString() + ". Minimal: " + minAmountToReceive.toString())

else if i.payments.size() != 1 then

throw("One attached payment expected")

else if !hasEnoughBalance then

suspendSuspicious()

else if pmtAssetId != assetIdA && pmtAssetId != assetIdB then

throwAssets()

else if pmtAmount < 10000000 then

throw("Only swap of 10.000000 or more tokens is allowed")

else if scale8.fraction(minAmountToReceive, pmtAmount) < exchangeRatioLimitMin ||

scale8.fraction(estimatedAmountToReceive, pmtAmount) > exchangeRatioLimitMax then

throw("Incorrect args and pmt ratio")

else {

let sendAssetId = if pmtAssetId == assetIdA then assetIdB else assetIdA


let amount = calculateSendAmount(estimatedAmountToReceive, minAmountToReceive, pmtAmount,
pmtAssetId)

let governanceReward = amount.fraction(feeGovernance, feeScale6)

let amountMinusFee = amount.fraction(feeScale6 - fee, feeScale6)


let (newBalanceA, newBalanceB) =

if pmtAssetId == assetIdA then

(balanceA + pmtAmount, balanceB - amountMinusFee - governanceReward)

else

(balanceA - amountMinusFee - governanceReward, balanceB + pmtAmount)

let dAppThresholdAmount = fraction(newBalanceA + newBalanceB, dAppThreshold, 2 *
dAppThresholdScale2)
```

```
if newBalanceA < dAppThresholdAmount || newBalanceB < dAppThresholdAmount then
throwThreshold(dAppThresholdAmount, newBalanceA, newBalanceB)
else if assetIdA == USDN && sendAssetId == assetIdA && newBalanceA <= stakedAmountUSDN then
throwInsufficientAvailableBalance(amountMinusFee, availableBalanceA, assetNameA)
else if assetIdB == USDN && sendAssetId == assetIdB && newBalanceB <= stakedAmountUSDN then
throwInsufficientAvailableBalance(amountMinusFee, availableBalanceB, assetNameB)
else
# successful execution result is updating information about actual balance and supply into the state and transfer tokens to the caller
[
IntegerEntry(kBalanceA, newBalanceA),
IntegerEntry(kBalanceB, newBalanceB),
IntegerEntry(kInvariant, invariantCalc(newBalanceA, newBalanceB)),
ScriptTransfer(i.caller, amountMinusFee, sendAssetId),
ScriptTransfer(govAddr, governanceReward, sendAssetId)
]
}
}

@Callable(i)
func shutdown() =
if !isActive then
throw("DApp is already suspended. Cause: " + this.getString(kCause).valueOrElse("the cause wasn't specified"))
else if ![adm1, adm2, adm3, admStartStop].containsElement(i.callerPublicKey) then
throwOnlyAdmin()
else
suspend("Paused by admin")

@Callable(i)
func activate() =
if isActive then
```

```
throwIsActive()
else if ![adm1, adm2, adm3, admStartStop].containsElement(i.callerPublicKey) then
throwOnlyAdmin()
else [
BooleanEntry(kActive, true),
DeleteEntry(kCause)
]

@Callable(i)
func takeIntoAccountExtraFunds(amountLeave: Int) = {
let uncountableA = accountBalanceWithStakedA - balanceA
let uncountableB = accountBalanceWithStakedB - balanceB
let amountEnrollA = uncountableA - if assetIdA == USDN then amountLeave else 0
let amountEnrollB = uncountableB - if assetIdB == USDN then amountLeave else 0
let invariantNew = invariantCalc(balanceA + amountEnrollA, balanceB + amountEnrollB)

if !isActive then
throwIsInactive()
else if i.caller != this then
throwOnlyAdmin()
else if amountLeave < 0 then
throw("Argument 'amountLeave' cannot be negative. Actual: " + amountLeave.toString())
else if uncountableA < 0 || uncountableB < 0 then
suspend("Enroll amount negative")
else if amountEnrollA < 0 || amountEnrollB < 0 then
throw("Too large amountLeave")
else
[
IntegerEntry(kInvariant, invariantNew),
IntegerEntry(kBalanceA, balanceA + amountEnrollA),
IntegerEntry(kBalanceB, balanceB + amountEnrollB),
```

```
IntegerEntry("last_income_"+strAssetIdA, amountEnrollA),
IntegerEntry("last_income_"+strAssetIdB, amountEnrollB)
]
}

@Verifier(tx)
func verify() = match tx {
case invoke: InvokeScriptTransaction =>
let callTakeIntoAccount = invoke.dApp == this && invoke.function == "takeIntoAccountExtraFunds"
let callStaking =
invoke.dApp == stakingAddress
&& (
(invoke.function == "lockNeutrino" && invoke.payments.size() == 1 && invoke.payments[0].assetId == USDN)
|| (invoke.function == "unlockNeutrino" && invoke.payments.size() == 0)
)
let signedByAdmin =
sigVerify(tx.bodyBytes, tx.proofs[0], adm1)
|| sigVerify(tx.bodyBytes, tx.proofs[0], adm2)
|| sigVerify(tx.bodyBytes, tx.proofs[0], adm3)
|| sigVerify(tx.bodyBytes, tx.proofs[0], admStaking)

(callTakeIntoAccount || callStaking) && signedByAdmin
case _ => {
let adm1Signed = if sigVerify(tx.bodyBytes, tx.proofs[0], adm1) then 1 else 0
let adm2Signed = if sigVerify(tx.bodyBytes, tx.proofs[1], adm2) then 1 else 0
let adm3Signed = if sigVerify(tx.bodyBytes, tx.proofs[2], adm3) then 1 else 0
adm1Signed + adm2Signed + adm3Signed >= 2
}
}
```

- Implementation of CPMM

Fairyproof's Analysis:

States can be set by a multi-signature address

Audit Result: No Issue

Code section is as follows:

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}

let version = "1.0.0"
let keyVersion = "version"
let keyActive = "active"
let keyAssetIdA = "A_asset_id"
let keyAssetIdB = "B_asset_id"
let keyBalanceA = "A_asset_balance"
let keyBalanceB = "B_asset_balance"
let keyBalanceInitA = "A_asset_init"
let keyBalanceInitB = "B_asset_init"
let keyShareAssetId = "share_asset_id"
let keyShareAssetSupply = "share_asset_supply"
let keyCommission = "commission"
let keyCommissionScaleDelimiter = "commission_scale_delimiter"
let keyCause = "shutdown_cause"
let keyFirstHarvest = "first_harvest"
let keyFirstHarvestHeight = "first_harvest_height"
let kShareLimit = "share_limit_on_first_harvest"
let kBasePeriod = "base_period"
let kPeriodLength = "period_length"
let kStartHeight = "start_height"
let kFirstHarvestHeight = "first_harvest_height"

let adminPubKey1 = base58'DXDY2itiEcYBtGkVLnkpHtDFyWQUkoLJz79uJ7ECbMrA'
let adminPubKey2 = base58'E6Wa1SGoktYcjHjsKrvjMiqJY3SWmGKcD8Q5L8kxSPS7'
let adminPubKey3 = base58'AZmWJtuy4GeVrMmJH4hfFBRApe1StvhJSk4jcbT6bArQ'
let adminPubKeyStartStop = base58'EtVkT6ed8GtbUiVVEqdmEqsp2J4qbb3rre2HFgxeVYdg'
let adminPubKeyStaking = base58'Czn4yoAuUZCVCLJDRfskn8URfkwpknwBTZDbs1wFrY7h'

let walletAddress = Address(base58'3P6J84oH51DzY6xk2mT5TheXRbrCwBMxonp')
let votingAddress = Address(base58'3PQZWxShKGRgBN1qoJw6B4s9YWS9FneZTPg')
let stakingAddress = Address(base58'3PNikM6yp4NqcSU8guxQtmR5onr2D4e8yTJ')

let basePeriod = votingAddress.getInteger(kBasePeriod).valueOrErrorMessage("Empty kBasePeriod") # 0
let startHeight = votingAddress.getInteger(kStartHeight).valueOrErrorMessage("Empty kStartHeight")
let periodLength = votingAddress.getInteger(kPeriodLength).valueOrErrorMessage("Empty kPeriodLength")
# 10102

let firstHarvestEndPeriod = basePeriod + (height-startHeight)/periodLength + 3
```

```
let USDN = base58'DG2xFkPdDwKUoBkzGAhQtLpSGzfXLiCYPEzeKH2Ad24p'
let stakingFeeInUSDN = 9 * assetInfo(USDN).value().minSponsoredFee.value() # sponsored fee for invoke called from scripted account

let isActive = this.getBooleanValue(keyActive)

let strAssetIdA = this.getStringValue(keyAssetIdA)
let strAssetIdB = this.getStringValue(keyAssetIdB)
let assetIdA = if strAssetIdA == "WAVES" then unit else strAssetIdA.fromBase58String()
let assetIdB = if strAssetIdB == "WAVES" then unit else strAssetIdB.fromBase58String()
let assetNameA = match assetIdA {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
let assetNameB = match assetIdB {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
let balanceA = this.getIntegerValue(keyBalanceA)
let balanceB = this.getIntegerValue(keyBalanceB)
let shareAssetId = this.getStringValue(keyShareAssetId).fromBase58String()
let shareAssetSupply = this.getIntegerValue(keyShareAssetSupply)

let commission = 3000 # commission/commissionScaleDelimiter = 0.003
let commissionGovernance = 1200 # commissionGovernance/commissionScaleDelimiter = 0.0012
let commissionScaleDelimiter = 1000000

let scaleValue3 = 1000
let scaleValue8 = 100000000
let slippageToleranceDelimiter = 1000
let scaleValue8Digits = 8

func accountBalance(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector => this.assetBalance(id)
case waves: Unit => this.wavesBalance().available
}

let stakedAmountUSDN = match stakingAddress.getInteger("rpd_balance" + USDN.toBase58String() + "" + this.toString()) {
case staked: Int => staked
case nothing: Unit => 0
}

let assetInitA = this.getIntegerValue(keyBalanceInitA)
let assetInitB = this.getIntegerValue(keyBalanceInitB)

let availableBalanceA = balanceA - if assetIdA == USDN then stakedAmountUSDN else 0
let availableBalanceB = balanceB - if assetIdB == USDN then stakedAmountUSDN else 0
let accountBalanceWithStakedA = accountBalance(assetIdA) + if assetIdA == USDN then stakedAmountUSDN else 0
let accountBalanceWithStakedB = accountBalance(assetIdB) + if assetIdB == USDN then
```

```
stakedAmountUSDN else 0

let hasEnoughBalance = accountBalanceWithStakedA >= balanceA && accountBalanceWithStakedB >= balanceB

func getAssetInfo(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector =>
let stringId = id.toBase58String()
let info = assetInfo(id).valueOrErrorMessage("Asset " + stringId + " doesn't exist")
(stringId, info.name, info.decimals)
case waves: Unit => ("WAVES", "WAVES", 8)
}

func getAssetInfoFromString(assetStr: String) = {
if assetStr == "WAVES" then ("WAVES", "WAVES", 8)
else {
let stringId = assetStr
let id = assetStr.fromBase58String()
let info = assetInfo(id).valueOrErrorMessage("Asset " + stringId + " doesn't exist")
(stringId, info.name, info.decimals)
}
}

func suspend(cause: String) = [
BooleanEntry(keyActive, false),
StringEntry(keyCause, cause)
]

func deductStakingFee(amount: Int, assetId: ByteVector|Unit) =
if assetId == USDN then {
let result = amount - stakingFeeInUSDN

if result <= 0 then
throw("Insufficient amount " + amount.toString() + " to deduct staking fee " + stakingFeeInUSDN.toString() + " USD-N")
else result
} else amount

func throwInsufficientAvailableBalance(amount: Int, available: Int, assetName: String) = throw("Insufficient DApp balance to pay " + amount.toString() + " " + assetName + " due to staking. Available: " + available.toString() + " " + assetName + ". Please contact support in Telegram: https://t.me/swopfisupport")
func throwInsufficientAvailableBalances(amountA: Int, amountB: Int) = throw("Insufficient DApp balance to pay " + amountA.toString() + " " + assetNameA + " and " + amountB.toString() + " " + assetNameB + " due to staking. Available: " + availableBalanceA.toString() + " " + assetNameA + " and " + availableBalanceB.toString() + " " + assetNameB + ". Please contact support in Telegram: https://t.me/swopfisupport")

@Callable(i)
func init(firstHarvest: Boolean) = {
let (pmtAmountA, pmtAssetIdA) = (i.payments[0].amount, i.payments[0].assetId)
let (pmtAmountB, pmtAssetIdB) = (i.payments[1].amount, i.payments[1].assetId)
```

```
let (pmtStrAssetIdA, pmtAssetNameA, pmtDecimalsA) = getAssetInfo(pmtAssetIdA)
let (pmtStrAssetIdB, pmtAssetNameB, pmtDecimalsB) = getAssetInfo(pmtAssetIdB)

if ![adminPubKey1, adminPubKey2, adminPubKey3,
adminPubKeyStaking].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else if this.getBoolean(keyActive).isDefined() then
throw("DApp is already active")
else if pmtAssetIdA == pmtAssetIdB then
throw("Assets must be different")
else {
let shareName = "s" + pmtAssetNameA.take(7) + "_" + pmtAssetNameB.take(7)
let shareDescription = "ShareToken of SwopFi protocol for " + pmtAssetNameA + " and " + pmtAssetNameB
+ " at address " + this.toString()

let shareDecimals = (pmtDecimalsA + pmtDecimalsB) / 2
let shareInitialSupply = fraction(
pow(pmtAmountA, pmtDecimalsA, 5, 1, pmtDecimalsA, HALFDOWN),
pow(pmtAmountB, pmtDecimalsB, 5, 1, pmtDecimalsB, HALFDOWN),
pow(10, 0, shareDecimals, 0, 0, HALFDOWN)
)
let shareIssue = Issue(shareName, shareDescription, shareInitialSupply, shareDecimals, true)
let shareIssueId = shareIssue.calculateAssetId()
let baseEntry = [
StringEntry(keyVersion, version),
BooleanEntry(keyActive, true),
StringEntry(keyAssetIdA, pmtStrAssetIdA),
StringEntry(keyAssetIdB, pmtStrAssetIdB),
IntegerEntry(keyBalanceA, pmtAmountA),
IntegerEntry(keyBalanceB, pmtAmountB),
IntegerEntry(keyCommission, commission),
IntegerEntry(keyCommissionScaleDelimiter, commissionScaleDelimiter),
shareIssue,
StringEntry(keyShareAssetId, shareIssueId.toBase58String()),
IntegerEntry(keyShareAssetSupply, shareInitialSupply),
ScriptTransfer(i.caller, shareInitialSupply, shareIssueId)
]
if (firstHarvest) then {
baseEntry ++ [
BooleanEntry(keyFirstHarvest, firstHarvest),
IntegerEntry(keyFirstHarvestHeight, startHeight + firstHarvestEndPeriod * periodLength)
]
} else {
baseEntry
}
}
}
```

```
@Callable(i)
func initWithInitRatio(amtAssetA: Int, amtAssetB: Int, strAssetIdA: String, strAssetIdB: String, firstHarvest:
Boolean) = {
let (pmtStrAssetIdA, pmtAssetNameA, pmtDecimalsA) = getAssetInfoFromString(strAssetIdA)
let (pmtStrAssetIdB, pmtAssetNameB, pmtDecimalsB) = getAssetInfoFromString(strAssetIdB)

if ![adminPubKey1, adminPubKey2, adminPubKey3,
adminPubKeyStaking].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else if this.getBoolean(keyActive).isDefined() then
throw("DApp is already active")
else if strAssetIdA == strAssetIdB then
throw("Assets must be different")
else {
let shareName = "s" + pmtAssetNameA.take(7) + "_" + pmtAssetNameB.take(7)
let shareDescription = "ShareToken of SwopFi protocol for " + pmtAssetNameA + " and " + pmtAssetNameB
+ " at address " + this.toString()

let shareDecimals = (pmtDecimalsA + pmtDecimalsB) / 2
let shareInitialSupply = 0
let shareIssue = Issue(shareName, shareDescription, shareInitialSupply, shareDecimals, true)
let shareIssueId = shareIssue.calculateAssetId()
let baseEntry = [
StringEntry(keyVersion, version),
BooleanEntry(keyActive, true),
StringEntry(keyAssetIdA, pmtStrAssetIdA),
StringEntry(keyAssetIdB, pmtStrAssetIdB),
IntegerEntry(keyBalanceInitA, amtAssetA),
IntegerEntry(keyBalanceInitB, amtAssetB),
IntegerEntry(keyBalanceA, 0),
IntegerEntry(keyBalanceB, 0),
IntegerEntry(keyCommission, commission),
IntegerEntry(keyCommissionScaleDelimiter, commissionScaleDelimiter),
shareIssue,
StringEntry(keyShareAssetId, shareIssueId.toBase58String()),
IntegerEntry(keyShareAssetSupply, shareInitialSupply)
]

if (firstHarvest) then {
baseEntry ++ [
BooleanEntry(keyFirstHarvest, firstHarvest),
IntegerEntry(keyFirstHarvestHeight, startHeight + firstHarvestEndPeriod * periodLength)
]
} else {
baseEntry
}
}
}
```

```
@Callable(i)
func keepLimitForFirstHarvest(shareLimit: Int) = {
if !isActive then
throw("DApp is inactive at this moment")
else if ![adminPubKey1, adminPubKey2, adminPubKey3,
adminPubKeyStaking].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else
[
IntegerEntry(kShareLimit, shareLimit)
]
}

@Callable(i)
func replenishWithTwoTokens(slippageTolerance: Int) = {
let pmtAssetIdA = i.payments[0].assetId
let pmtAssetIdB = i.payments[1].assetId
```

- Checkpoint1: block for accounting the cost of commissions for staking operations

```
let pmtAmountA = deductStakingFee(i.payments[0].amount, pmtAssetIdA)
let pmtAmountB = deductStakingFee(i.payments[1].amount, pmtAssetIdB)

if (balanceA == 0 && balanceB == 0) then {
let (pmtStrAssetIdA, pmtAssetNameA, pmtDecimalsA) = getAssetInfo(pmtAssetIdA)
let (pmtStrAssetIdB, pmtAssetNameB, pmtDecimalsB) = getAssetInfo(pmtAssetIdB)
let tokenRatio = fraction(assetInitA, scaleValue8, pmtAmountA)
.fraction(scaleValue3, assetInitB.fraction(scaleValue8, pmtAmountB))

if pmtAssetIdA == pmtAssetIdB then
throw("Assets must be different")
else {
let shareDecimals = (pmtDecimalsA + pmtDecimalsB) / 2
let shareInitialSupply = fraction(
pow(pmtAmountA, pmtDecimalsA, 5, 1, pmtDecimalsA, HALFDOWN),
pow(pmtAmountB, pmtDecimalsB, 5, 1, pmtDecimalsB, HALFDOWN),
pow(10, 0, shareDecimals, 0, 0, HALFDOWN)
)

if !isActive then
throw("DApp is inactive at this moment")
else if slippageTolerance < 0 || slippageTolerance > slippageToleranceDelimiter then
throw("Slippage tolerance must be between 0 and " + slippageToleranceDelimiter.toString() + " inclusively.
Actual: " + slippageTolerance.toString())
else if i.payments.size() != 2 then
throw("Two attached assets expected")
else if tokenRatio < (scaleValue3 * (slippageToleranceDelimiter - slippageTolerance)) /
slippageToleranceDelimiter
```

```
|| tokenRatio > (scaleValue3 * (slippageToleranceDelimiter + slippageTolerance)) /
slippageToleranceDelimiter then
throw("Incorrect assets amount: amounts must have the contract ratio")
else if pmtAssetIdA != assetIdA || pmtAssetIdB != assetIdB then
throw("Incorrect assets attached. Expected: " + strAssetIdA + " and " + strAssetIdB)
else if shareInitialSupply == 0 then
throw("Too small amount to replenish")
else if !hasEnoughBalance then
suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else
[
Reissue(shareAssetId, shareInitialSupply, true),
IntegerEntry(keyBalanceA, pmtAmountA),
IntegerEntry(keyBalanceB, pmtAmountB),
IntegerEntry(keyShareAssetSupply, shareInitialSupply),
ScriptTransfer(i.caller, shareInitialSupply, shareAssetId)
]
}
} else

#fraction should be equal 1(multiple by 1000) if depositor replenish with proportion according to actual
#price

let tokenRatio = fraction(balanceA, scaleValue8, pmtAmountA)
.fraction(scaleValue3, balanceB.fraction(scaleValue8, pmtAmountB))

let ratioShareTokensInA = fraction(pmtAmountA, scaleValue8, balanceA)
let ratioShareTokensInB = fraction(pmtAmountB, scaleValue8, balanceB)
let shareTokenToPayAmount = min([ratioShareTokensInA, ratioShareTokensInB]).fraction(shareAssetSupply,
scaleValue8)

if !isActive then
throw("DApp is inactive at this moment")
else if slippageTolerance < 0 || slippageTolerance > slippageToleranceDelimiter then
throw("Slippage tolerance must be between 0 and " + slippageToleranceDelimiter.toString() + " inclusively.
Actual: " + slippageTolerance.toString())
else if i.payments.size() != 2 then
throw("Two attached assets expected")
else if pmtAssetIdA != assetIdA || pmtAssetIdB != assetIdB then
throw("Incorrect assets attached. Expected: " + strAssetIdA + " and " + strAssetIdB)
else if tokenRatio < (scaleValue3 * (slippageToleranceDelimiter - slippageTolerance)) /
slippageToleranceDelimiter
|| tokenRatio > (scaleValue3 * (slippageToleranceDelimiter + slippageTolerance)) /
slippageToleranceDelimiter then
throw("Incorrect assets amount: amounts must have the contract ratio")
else if shareTokenToPayAmount == 0 then
throw("Too small amount to replenish")
```

```
else if !hasEnoughBalance then
suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else [
IntegerEntry(keyBalanceA, balanceA + pmtAmountA),
IntegerEntry(keyBalanceB, balanceB + pmtAmountB),
IntegerEntry(keyShareAssetSupply, shareAssetSupply + shareTokenToPayAmount),
Reissue(shareAssetId, shareTokenToPayAmount, true),
ScriptTransfer(i.caller, shareTokenToPayAmount, shareAssetId)
]
}

@Callable(i)
func withdraw() = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)
```

- Checkpoint2: block for accounting the cost of commissions for staking operations

```
let amountToPayA = pmtAmount.fraction(balanceA, shareAssetSupply).deductStakingFee(assetIdA)
let amountToPayB = pmtAmount.fraction(balanceB, shareAssetSupply).deductStakingFee(assetIdB)

if !isActive then
throw("DApp is inactive at this moment")
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if pmtAssetId != shareAssetId then
throw("Incorrect asset attached. Expected: " + shareAssetId.toBase58String())
else if !hasEnoughBalance then
suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else if amountToPayA > availableBalanceA || amountToPayB > availableBalanceB then
throwInsufficientAvailableBalances(amountToPayA, amountToPayB)
else [
IntegerEntry(keyBalanceA, balanceA - amountToPayA),
IntegerEntry(keyBalanceB, balanceB - amountToPayB),
IntegerEntry(keyShareAssetSupply, shareAssetSupply - pmtAmount),
Burn(shareAssetId, pmtAmount),
ScriptTransfer(i.caller, amountToPayA, assetIdA),
ScriptTransfer(i.caller, amountToPayB, assetIdB)
]
}

@Callable(i)
func exchange(minAmountToReceive: Int) = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)
```

```
func calculateFees(tokenFrom: Int, tokenTo: Int) = {
let amountWithoutFee = fraction(tokenTo, pmtAmount, pmtAmount + tokenFrom)
let amountWithFee = fraction(amountWithoutFee, commissionScaleDelimiter - commission,
commissionScaleDelimiter)
let governanceReward = fraction(amountWithoutFee, commissionGovernance, commissionScaleDelimiter)

if amountWithFee < minAmountToReceive then
throw("Calculated amount to receive " + amountWithFee.toString() + " is less than specified minimum " +
minAmountToReceive.toString())
else
(amountWithoutFee, amountWithFee, governanceReward)
}

if !isActive then
throw("DApp is inactive at this moment")
else if balanceA == 0 || balanceB == 0 then
throw("Can't exchange with zero balance")
else if minAmountToReceive <= 0 then
throw("Minimal amount to receive must be positive. Actual: " + minAmountToReceive.toString())
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if !hasEnoughBalance then
suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else if pmtAssetId == assetIdA then {
let assetIdSend = assetIdB

let (amountWithoutFee, amountWithFee, governanceReward) = calculateFees(balanceA, balanceB)

let newBalanceA = balanceA + pmtAmount
let newBalanceB = balanceB - amountWithFee - governanceReward
```

- Checkpoint3: successful execution result is updating information about actual balance and supply into the state and transfer tokens to the caller

```
if (assetIdA == USDN && newBalanceA <= stakedAmountUSDN) || (assetIdB == USDN && newBalanceB <=
stakedAmountUSDN) then
throwInsufficientAvailableBalance(amountWithFee, availableBalanceB, assetNameB)
else
[
IntegerEntry(keyBalanceA, newBalanceA),
IntegerEntry(keyBalanceB, newBalanceB),
ScriptTransfer(i.caller, amountWithFee, assetIdSend),
ScriptTransfer(walletAddress, governanceReward, assetIdSend)
]
} else if pmtAssetId == assetIdB then {
```

```
let assetIdSend = assetIdA
let (amountWithoutFee, amountWithFee, governanceReward) = calculateFees(balanceB, balanceA)

let newBalanceA = balanceA - amountWithFee - governanceReward
let newBalanceB = balanceB + pmtAmount

# successful execution result is updating information about actual balance and supply into the state and transfer tokens to the caller

if (assetIdA == USDN && newBalanceA <= stakedAmountUSDN) || (assetIdB == USDN && newBalanceB <= stakedAmountUSDN) then
throwInsufficientAvailableBalance(amountWithFee, availableBalanceA, assetNameA)
else
[
IntegerEntry(keyBalanceA, newBalanceA),
IntegerEntry(keyBalanceB, newBalanceB),
ScriptTransfer(i.caller, amountWithFee, assetIdSend),
ScriptTransfer(walletAddress, governanceReward, assetIdSend)
]
} else
throw("Incorrect asset attached. Expected: " + strAssetIdA + " or " + strAssetIdB)
}

@Callable(i)
func shutdown() =
if !isActive then
throw("DApp is already suspended. Cause: " + this.getString(keyCause).valueOrElse("the cause wasn't specified"))
else if ![adminPubKey1, adminPubKey2, adminPubKey3, adminPubKeyStartStop].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else
suspend("Paused by admin")

@Callable(i)
func activate() =
if isActive then
throw("DApp is already active")
else if ![adminPubKey1, adminPubKey2, adminPubKey3, adminPubKeyStartStop].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else [
BooleanEntry(keyActive, true),
DeleteEntry(keyCause)
]

@Callable(i)
func takeIntoAccountExtraFunds(amountLeave: Int) = {
let uncountableAmountEnrollAssetA = accountBalanceWithStakedA - balanceA
let uncountableAmountEnrollAssetB = accountBalanceWithStakedB - balanceB
```

```
let amountEnrollA = uncountableAmountEnrollAssetA - if assetIdA == USDN then amountLeave else 0
let amountEnrollB = uncountableAmountEnrollAssetB - if assetIdB == USDN then amountLeave else 0

if !isActive then
throw("DApp is inactive at this moment")
else if i.caller != this then
throw("Only the DApp itself can call this function")
else if amountLeave < 0 then
throw("Argument 'amountLeave' cannot be negative. Actual: " + amountLeave.toString())
else if uncountableAmountEnrollAssetA < 0 || uncountableAmountEnrollAssetB < 0 then
suspend("Enroll amount negative")
else if amountEnrollA < 0 || amountEnrollB < 0 then
throw("Too large amountLeave")
else
[
IntegerEntry(keyBalanceA, balanceA + amountEnrollA),
IntegerEntry(keyBalanceB, balanceB + amountEnrollB),
IntegerEntry("last_income"+strAssetIdA, amountEnrollA),
IntegerEntry("last_income"+strAssetIdB, amountEnrollB)
]
}

@Verifier(tx)
func verify() = match tx {
case invoke: InvokeScriptTransaction =>
let callTakeIntoAccount = invoke.dApp == this && invoke.function == "takeIntoAccountExtraFunds"
let callStaking =
invoke.dApp == stakingAddress
&& (
(invoke.function == "lockNeutrino" && invoke.payments.size() == 1 && invoke.payments[0].assetId == USDN)
|| (invoke.function == "unlockNeutrino" && invoke.payments.size() == 0)
)
let signedByAdmin =
sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey1)
|| sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey2)
|| sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey3)
|| sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKeyStaking)

(callTakeIntoAccount || callStaking) && signedByAdmin
case _ => {
let adminPubKey1Signed = if sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey1) then 1 else 0
let adminPubKey2Signed = if sigVerify(tx.bodyBytes, tx.proofs[1], adminPubKey2) then 1 else 0
let adminPubKey3Signed = if sigVerify(tx.bodyBytes, tx.proofs[2], adminPubKey3) then 1 else 0
adminPubKey1Signed + adminPubKey2Signed + adminPubKey3Signed >= 2
}
}
```

- Early Bird Program

Fairyproof's Analysis:

The following code implements the early bird program.

Only liquidity providers can get rewards. The total reward amount is (100000000000000 / 10^8)

Audit Result: No Issue

Code section is as follows:

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}


let keyActivateHeight = "activate_height"
let keyFinishHeight = "finish_height"
let activateHeight = this.getIntegerValue(keyActivateHeight)
let finishHeight = this.getIntegerValue(keyFinishHeight)
let totalShareSWOP = 100000000000000 # 1m with 8 digits
let SWOP = base58'Ehie5xYpeN8op1Cctc6aGUrqx8jq3jtf1DSjXDbfm7aT'
let keyUserSWOPClaimedAmount = "SWOP_claimed_amount"
let keyUserSWOPLastClaimedAmount = "SWOP_last_claimed_amount"
let adminPubKey1 = base58'DXDY2itiEcYBtGkVLnkpHtDFyWQUkoLJz79uJ7ECbMrA'
let adminPubKey2 = base58'E6Wa1SGoktYcjHjsKrvjMiqJY3SWmGKcD8Q5L8kxSPS7'
let adminPubKey3 = base58'AZmWJtuy4GeVrMmJH4hfFBRApe1StvhJSk4jcbT6bArQ'
```

- Checkpoint1: Get Caller's Share

```
func getCallerShare(caller:Address) = {
let callerShare = this.getInteger("share_" + caller.toString())
let callerShareAmount = match callerShare {
case share: Int => share
case share: Unit => throw("Only early liquidity providers can call this function")
}
callerShareAmount
}
```

- Checkpoint2: Get Claimed Amount

```
func getClaimedAmount(caller:Address) = {
let callerWithdrawn = this.getInteger(caller.toString()+ keyUserSWOPClaimedAmount)
let callerWithdrawnAmount = match callerWithdrawn {
case share: Int => share
case share: Unit => 0
}
```

```
callerWithdrawnAmount
}

@Callable(i)
func claimSWOP() = {
let blockDuration = finishHeight - activateHeight
let currentDuration = if height < finishHeight then height else finishHeight
let userShare = getCallerShare(i.caller)
let userClaimedAmount = getClaimedAmount(i.caller) # already withdrawn amount
let claimAmount = (currentDuration-activateHeight).fraction(userShare,blockDuration) -
userClaimedAmount
let userClaimedAmountNew = userClaimedAmount + claimAmount
[
ScriptTransfer(i.caller, claimAmount, SWOP),
IntegerEntry(i.caller.toString() + keyUserSWOPClaimedAmount, userClaimedAmountNew),
IntegerEntry(i.caller.toString() + keyUserSWOPLastClaimedAmount, claimAmount)
]
}

@Verifier(tx)
func verify() = match tx {
case _ => {
let adminPubKey1Signed = if sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey1) then 1 else 0
let adminPubKey2Signed = if sigVerify(tx.bodyBytes, tx.proofs[1], adminPubKey2) then 1 else 0
let adminPubKey3Signed = if sigVerify(tx.bodyBytes, tx.proofs[2], adminPubKey3) then 1 else 0
adminPubKey1Signed + adminPubKey2Signed + adminPubKey3Signed >= 2
}
}
```

- Swap Mechanism

Fairyproof's Analysis:

Swap Among NSBT, USDN and SWOP

Audit Result: No Issue

Code section is as follows:

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}


let version = "1.0.0"
```

```
let keyVersion = "version"
let keyActive = "active"
let keyAssetIdA = "A_asset_id"
let keyAssetIdB = "B_asset_id"
let keyBalanceA = "A_asset_balance"
let keyBalanceB = "B_asset_balance"
let keyShareAssetId = "share_asset_id"
let keyShareAssetSupply = "share_asset_supply"
let keyCommission = "commission"
let keyCommissionScaleDelimiter = "commission_scale_delimiter"
let keyCause = "shutdown_cause"

let adminPubKey1 = base58'DXDY2itiEcYBtGkVLnkpHtDFyWQUkoLJz79uJ7ECbMrA'
let adminPubKey2 = base58'E6Wa1SGoktYcjHjsKrvjMiqJY3SWmGKcD8Q5L8kxSPS7'
let adminPubKey3 = base58'AZmWJtuy4GeVrMmJH4hfFBRApe1StvhJSk4jcbT6bArQ'
let adminPubKeyStartStop = base58'EtVkT6ed8GtbUiVVEqdmEqsp2J4qbb3rre2HFgxeVYdg'
let adminPubKeyStaking = base58'Czn4yoAuUZCVCLJDRfskn8URfkwpknwBTZDbs1wFrY7h'

let governanceAddress = Address(base58'3P6J84oH51DzY6xk2mT5TheXRbrCwBMxonp')
let stakingAddressUSDN = Address(base58'3PNikM6yp4NqcSU8guxQtmR5onr2D4e8yTJ')

let USDN = base58'DG2xFkPdDwKUoBkzGAhQtLpSGzfXLiCYPEzeKH2Ad24p'
let NSBT = base58'6nSpVyNH7yM69eg446wrQR94ipbbcmZMU1ENPwanC97g'
let SWOP = base58'Ehie5xYpeN8op1Cctc6aGUrqx8jq3jtf1DSjXDbfm7aT'
let swopUSDNtoWAVES = Address(base58'3PHaNgomBkrvEL2QnuJarQVJa71wjw9qiqG')
let swopUSDNtoNSBT = Address(base58'3P2V63Xd6BviDkeMzxhUw2SJyojByRz8a8m')

let stakingFeeInUSDN = 9 * assetInfo(USDN).value().minSponsoredFee.value() # sponsored fee for invoke called from scripted account

let isActive = this.getBooleanValue(keyActive)

let strAssetIdA = this.getStringValue(keyAssetIdA)
let strAssetIdB = this.getStringValue(keyAssetIdB)
let assetIdA = if strAssetIdA == "WAVES" then unit else strAssetIdA.fromBase58String()
let assetIdB = if strAssetIdB == "WAVES" then unit else strAssetIdB.fromBase58String()
let assetNameA = match assetIdA {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
let assetNameB = match assetIdB {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
let balanceA = this.getIntegerValue(keyBalanceA)
let balanceB = this.getIntegerValue(keyBalanceB)
let shareAssetId = this.getStringValue(keyShareAssetId).fromBase58String()
let shareAssetSupply = this.getIntegerValue(keyShareAssetSupply)
```

```
let commission = 3000 # commission/commissionScaleDelimiter = 0.003
let commissionGovernance = 1200 # commissionGovernance/commissionScaleDelimiter = 0.0012
let commissionScaleDelimiter = 1000000

let scaleValue3 = 1000
let scaleValue8 = 100000000
let slippageToleranceDelimiter = 1000
let scaleValue8Digits = 8

func accountBalance(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector => this.assetBalance(id)
case waves: Unit => this.wavesBalance().available
}

func stakedAmount(stakingAddress: Address,asset: String) = {
let stakedAmountCalculated = match stakingAddress.getInteger("rpd_balance" + asset + "" + this.toString()) {
case staked: Int => staked
case nothing: Unit => 0
}
stakedAmountCalculated
}

let stakedAmountA = stakedAmount(stakingAddressUSDN,strAssetIdA)
let stakedAmountB = stakedAmount(stakingAddressUSDN,strAssetIdB)

let availableBalanceA = balanceA - stakedAmountA
let availableBalanceB = balanceB - stakedAmountB
let accountBalanceWithStakedA = accountBalance(assetIdA) + stakedAmountA
let accountBalanceWithStakedB = accountBalance(assetIdB) + stakedAmountB

let hasEnoughBalance = accountBalanceWithStakedA >= balanceA && accountBalanceWithStakedB >=
balanceB

func getAssetInfo(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector =>
let stringId = id.toBase58String()
let info = assetInfo(id).valueOrErrorMessage("Asset " + stringId + " doesn't exist")
(stringId, info.name, info.decimals)
case waves: Unit => ("WAVES", "WAVES", 8)
}

func suspend(cause: String) = [
BooleanEntry(keyActive, false),
StringEntry(keyCause, cause)
]
```

- Checkpoint1: calculate amount of staked tokens

```
func stakedTokenCount() = {
let isStakedA = if stakedAmountA > 0 then 1 else 0
let isStakedB = if stakedAmountB > 0 then 1 else 0
isStakedA + isStakedB
}

func deductStakingFee(amount: Int, assetId: ByteVector|Unit) =

if assetId == USDN then {
let result = amount - stakedTokenCount()*stakingFeeInUSDN
if result <= 0 then
throw("Insufficient amount " + amount.toString() + " to deduct staking fee " + stakingFeeInUSDN.toString() +
" USD-N")
else result
} else amount

func throwInsufficientAvailableBalance(amount: Int, available: Int, assetName: String) = throw("Insufficient
DApp balance to pay " + amount.toString() + " " + assetName + " due to staking. Available: " +
available.toString() + " " + assetName + ". Please contact support in Telegram: https://t.me/swopfisupport")
func throwInsufficientAvailableBalances(amountA: Int, amountB: Int) = throw("Insufficient DApp balance to
pay " + amountA.toString() + " " + assetNameA + " and " + amountB.toString() + " " + assetNameB + " due to
staking. Available: " + availableBalanceA.toString() + " " + assetNameA + " and " +
availableBalanceB.toString() + " " + assetNameB + ". Please contact support in Telegram: https://t.me/swopfi
support")

@Callable(i)
func init() = {
let (pmtAmountA, pmtAssetIdA) = (i.payments[0].amount, i.payments[0].assetId)
let (pmtAmountB, pmtAssetIdB) = (i.payments[1].amount, i.payments[1].assetId)
let (pmtStrAssetIdA, pmtAssetNameA, pmtDecimalsA) = getAssetInfo(pmtAssetIdA)
let (pmtStrAssetIdB, pmtAssetNameB, pmtDecimalsB) = getAssetInfo(pmtAssetIdB)

if this.getBoolean(keyActive).isDefined() then
throw("DApp is already active")
else if pmtAssetIdA == pmtAssetIdB then
throw("Assets must be different")
else {
let shareName = "s" + pmtAssetNameA.take(7) + "_" + pmtAssetNameB.take(7)
let shareDescription = "ShareToken of SwopFi protocol for " + pmtAssetNameA + " and " + pmtAssetNameB
+ " at address " + this.toString()

let shareDecimals = (pmtDecimalsA + pmtDecimalsB) / 2
let shareInitialSupply = fraction(
pow(pmtAmountA, pmtDecimalsA, 5, 1, pmtDecimalsA, HALFDOWN),
pow(pmtAmountB, pmtDecimalsB, 5, 1, pmtDecimalsB, HALFDOWN),
pow(10, 0, shareDecimals, 0, 0, HALFDOWN)
)
let shareIssue = Issue(shareName, shareDescription, shareInitialSupply, shareDecimals, true)
```

```
let shareIssueId = shareIssue.calculateAssetId()
[
StringEntry(keyVersion, version),
BooleanEntry(keyActive, true),
StringEntry(keyAssetIdA, pmtStrAssetIdA),
StringEntry(keyAssetIdB, pmtStrAssetIdB),
IntegerEntry(keyBalanceA, pmtAmountA),
IntegerEntry(keyBalanceB, pmtAmountB),
IntegerEntry(keyCommission, commission),
IntegerEntry(keyCommissionScaleDelimiter, commissionScaleDelimiter),
shareIssue,
StringEntry(keyShareAssetId, shareIssueId.toBase58String()),
IntegerEntry(keyShareAssetSupply, shareInitialSupply),
ScriptTransfer(i.caller, shareInitialSupply, shareIssueId)
]
}
}

@Callable(i)
func replenishWithTwoTokens(slippageTolerance: Int) = {
let pmtAssetIdA = i.payments[0].assetId
let pmtAssetIdB = i.payments[1].assetId
```

- Checkpoint2: block for accounting the cost of commissions for staking operations

```
let pmtAmountA = deductStakingFee(i.payments[0].amount, pmtAssetIdA)
let pmtAmountB = deductStakingFee(i.payments[1].amount, pmtAssetIdB)
```

- Checkpoint3: fraction should be equal 1(multiple by 1000) if depositor replenish with proportion according to actual price

```
let tokenRatio = fraction(balanceA, scaleValue8, pmtAmountA)
.fraction(scaleValue3, balanceB.fraction(scaleValue8, pmtAmountB))
let ratioShareTokensInA = fraction(pmtAmountA, scaleValue8, balanceA)
let ratioShareTokensInB = fraction(pmtAmountB, scaleValue8, balanceB)
let shareTokenToPayAmount = min([ratioShareTokensInA, ratioShareTokensInB]).fraction(shareAssetSupply,
scaleValue8)

if !isActive then
throw("DApp is inactive at this moment")
else if slippageTolerance < 0 || slippageTolerance > slippageToleranceDelimiter then
throw("Slippage tolerance must be between 0 and " + slippageToleranceDelimiter.toString() + " inclusively.
Actual: " + slippageTolerance.toString())
else if i.payments.size() != 2 then
throw("Two attached assets expected")
else if pmtAssetIdA != assetIdA || pmtAssetIdB != assetIdB then
throw("Incorrect assets attached. Expected: " + strAssetIdA + " and " + strAssetIdB)
```

```
else if tokenRatio < (scaleValue3 * (slippageToleranceDelimiter - slippageTolerance)) /
slippageToleranceDelimiter
|| tokenRatio > (scaleValue3 * (slippageToleranceDelimiter + slippageTolerance)) /
slippageToleranceDelimiter then
throw("Incorrect assets amount: amounts must have the contract ratio")
else if shareTokenToPayAmount == 0 then
throw("Too small amount to replenish")
else if !hasEnoughBalance then
[
ScriptTransfer(i.caller, pmtAmountA, pmtAssetIdA),
ScriptTransfer(i.caller, pmtAmountB, pmtAssetIdB)
] ++ suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else [
IntegerEntry(keyBalanceA, balanceA + pmtAmountA),
IntegerEntry(keyBalanceB, balanceB + pmtAmountB),
IntegerEntry(keyShareAssetSupply, shareAssetSupply + shareTokenToPayAmount),
Reissue(shareAssetId, shareTokenToPayAmount, true),
ScriptTransfer(i.caller, shareTokenToPayAmount, shareAssetId)
]
}
```

- Checkpoint4: Withdraw Function

```
@Callable(i)
func withdraw() = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)


let amountToPayA = pmtAmount.fraction(balanceA, shareAssetSupply).deductStakingFee(assetIdA)
let amountToPayB = pmtAmount.fraction(balanceB, shareAssetSupply).deductStakingFee(assetIdB)
```

- Checkpoint5: Check States

```
if !isActive then
throw("DApp is inactive at this moment")
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if pmtAssetId != shareAssetId then
throw("Incorrect asset attached. Expected: " + shareAssetId.toBase58String())
else if !hasEnoughBalance then
[
ScriptTransfer(i.caller, pmtAmount, pmtAssetId)
] ++ suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
```

```
else if amountToPayA > availableBalanceA || amountToPayB > availableBalanceB then
throwInsufficientAvailableBalances(amountToPayA, amountToPayB)
else [
IntegerEntry(keyBalanceA, balanceA - amountToPayA),
IntegerEntry(keyBalanceB, balanceB - amountToPayB),
IntegerEntry(keyShareAssetSupply, shareAssetSupply - pmtAmount),
Burn(shareAssetId, pmtAmount),
ScriptTransfer(i.caller, amountToPayA, assetIdA),
ScriptTransfer(i.caller, amountToPayB, assetIdB)
]
}

@Callable(i)
func exchange(minAmountToReceive: Int) = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)
```

- Checkpoint6: Calculate Fees

```
func calculateFees(tokenFrom: Int, tokenTo: Int) = {
let amountWithoutFee = fraction(tokenTo, pmtAmount, pmtAmount + tokenFrom)
let amountWithFee = fraction(amountWithoutFee, commissionScaleDelimiter - commission,
commissionScaleDelimiter)
let governanceReward = fraction(amountWithoutFee, commissionGovernance, commissionScaleDelimiter)

if amountWithFee < minAmountToReceive then
throw("Calculated amount to receive " + amountWithFee.toString() + " is less than specified minimum " +
minAmountToReceive.toString())
else
(amountWithoutFee, amountWithFee, governanceReward)
}
```

- Checkpoint7: Check States

```
if !isActive then
throw("DApp is inactive at this moment")
else if minAmountToReceive <= 0 then
throw("Minimal amount to receive must be positive. Actual: " + minAmountToReceive.toString())
else if i.payments.size() != 1 then
throw("One attached payment expected")
else if !hasEnoughBalance then
[
ScriptTransfer(i.caller, pmtAmount, pmtAssetId)
] ++ suspend("Suspicious state. Actual balances: " + balanceA.toString() + " " + assetNameA + ", " +
balanceB.toString() + " " + assetNameB + ". State: " + accountBalance(assetIdA).toString() + " " + assetNameA
+ ", " + accountBalance(assetIdB).toString() + " " + assetNameB)
else if pmtAssetId == assetIdA then {
let assetIdSend = assetIdB
let (amountWithoutFee, amountWithFee, governanceReward) = calculateFees(balanceA, balanceB)
```

```
let newBalanceA = balanceA + pmtAmount
let newBalanceB = balanceB - amountWithFee - governanceReward

if newBalanceA <= stakedAmountA || newBalanceB <= stakedAmountB then
throwInsufficientAvailableBalance(amountWithFee, availableBalanceB, assetNameB)
else
[
IntegerEntry(keyBalanceA, newBalanceA),
IntegerEntry(keyBalanceB, newBalanceB),
ScriptTransfer(i.caller, amountWithFee, assetIdSend),
ScriptTransfer(governanceAddress, governanceReward, assetIdSend)
]
} else if pmtAssetId == assetIdB then {
let assetIdSend = assetIdA
let (amountWithoutFee, amountWithFee, governanceReward) = calculateFees(balanceB, balanceA)

let newBalanceA = balanceA - amountWithFee - governanceReward
let newBalanceB = balanceB + pmtAmount

if newBalanceA <= stakedAmountA || newBalanceB <= stakedAmountB then
throwInsufficientAvailableBalance(amountWithFee, availableBalanceA, assetNameA)
else
[
IntegerEntry(keyBalanceA, newBalanceA),
IntegerEntry(keyBalanceB, newBalanceB),
ScriptTransfer(i.caller, amountWithFee, assetIdSend),
ScriptTransfer(governanceAddress, governanceReward, assetIdSend)
]
} else
throw("Incorrect asset attached. Expected: " + strAssetIdA + " or " + strAssetIdB)
}
```

- Checkpoint8: Admin Operation

```
@Callable(i)
func shutdown() =
if !isActive then
throw("DApp is already suspended. Cause: " + this.getString(keyCause).valueOrElse("the cause wasn't
specified"))
else if ![adminPubKey1, adminPubKey2, adminPubKey3,
adminPubKeyStartStop].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else
suspend("Paused by admin")
```

```
@Callable(i)
func activate() =
if isActive then
throw("DApp is already active")
else if ![adminPubKey1, adminPubKey2, adminPubKey3,
adminPubKeyStartStop].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else [
BooleanEntry(keyActive, true),
DeleteEntry(keyCause)
]

@Callable(i)
func takeIntoAccountExtraFunds(amountLeave: Int) = {
let uncountableAmountEnrollAssetA = accountBalanceWithStakedA - balanceA
let uncountableAmountEnrollAssetB = accountBalanceWithStakedB - balanceB
let amountEnrollA = uncountableAmountEnrollAssetA - if assetIdA == USDN then amountLeave else 0
let amountEnrollB = uncountableAmountEnrollAssetB - if assetIdB == USDN then amountLeave else 0

if !isActive then
throw("DApp is inactive at this moment")
else if i.caller != this then
throw("Only the DApp itself can call this function")
else if amountLeave < 0 then
throw("Argument 'amountLeave' cannot be negative. Actual: " + amountLeave.toString())
else if uncountableAmountEnrollAssetA < 0 || uncountableAmountEnrollAssetB < 0 then
suspend("Enroll amount negative")
else if amountEnrollA < 0 || amountEnrollB < 0 then
throw("Too large amountLeave")
else
[
IntegerEntry(keyBalanceA, balanceA + amountEnrollA),
IntegerEntry(keyBalanceB, balanceB + amountEnrollB),
IntegerEntry("last_income"+strAssetIdA, amountEnrollA),
IntegerEntry("last_income"+strAssetIdB, amountEnrollB)
]
}
```

- Checkpoint9: Verify Transaction

```
@Verifier(tx)
func verify() = match tx {
case invoke: InvokeScriptTransaction =>
(sigVerify(invoke.bodyBytes, invoke.proofs[0], adminPubKeyStaking)
&& invoke.dApp == stakingAddressUSDN)
||
(sigVerify(invoke.bodyBytes, invoke.proofs[0], adminPubKeyStaking)
```

```
&& invoke.function == "exchange"
&& ((
assetIdA == NSBT && assetIdB == USDN
&& invoke.dApp == swopUSDNtoWAVES
&& invoke.payments[0].assetId == unit
) ||
(assetIdA == NSBT && assetIdB == SWOP
&& (
(invoke.dApp == swopUSDNtoNSBT && invoke.payments[0].assetId == USDN)
||
(invoke.dApp == swopUSDNtoWAVES && invoke.payments[0].assetId == unit)
)))
)
||
(sigVerify(invoke.bodyBytes, invoke.proofs[0], adminPubKeyStaking)
&& invoke.dApp == this
&& invoke.function == "takeIntoAccountExtraFunds")

case _ => {
let adminPubKey1Signed = if sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey1) then 1 else 0
let adminPubKey2Signed = if sigVerify(tx.bodyBytes, tx.proofs[1], adminPubKey2) then 1 else 0
let adminPubKey3Signed = if sigVerify(tx.bodyBytes, tx.proofs[2], adminPubKey3) then 1 else 0
adminPubKey1Signed + adminPubKey2Signed + adminPubKey3Signed >= 2
}
}
```

- Farming

Fairyproof's Analysis:

Implementation of Farming

Audit Result: No Issue

Code section is as follows:

```
{-# STDLIB_VERSION 4 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}


let adminPubKey1 = base58'DXDY2itiEcYBtGkVLnkpHtDFyWQUkoLJz79uJ7ECbMrA'
let adminPubKey2 = base58'E6Wa1SGoktYcjHjsKrvjMiqJY3SWmGKcD8Q5L8kxSPS7'
let adminPubKey3 = base58'AZmWJtuy4GeVrMmJH4hfFBRApe1StvhJSk4jcbT6bArQ'
```

```
let keyShareTokensLocked = "total_share_tokens_locked" # with prefix(pool identity) get info about total share locked in this pool
let kShareLimit = "share_limit_on_first_harvest"
let keyActive = "active"
let keyCause = "shutdown_cause"
let keyRewardPoolFractionCurrent = "current_pool_fraction_reward"
let keyRewardPoolFractionPrevious = "previous_pool_fraction_reward"
let keyHeightPoolFraction = "pool_reward_update_height"
let keyTotalRewardPerBlockCurrent = "total_reward_per_block_current"
let keyTotalRewardPerBlockPrevious = "total_reward_per_block_previous"
let keyRewardUpdateHeight = "reward_update_height"
let keyLastInterest = "last_interest"
let keyLastInterestHeight = "last_interest_height"
let keyUserShareTokensLocked = "share_tokens_locked"
let keyUserLastInterest = "last_interest"
let keySWOPid = "SWOP_id"
let keyUserSWOPClaimedAmount = "SWOP_claimed_amount"
let keyUserSWOPLastClaimedAmount = "SWOP_last_claimed_amount"
let keyAvailableSWOP = "available_SWOP"
let keyFarmingStartHeight = "farming_start_height"
let keyAPY = "apy"
let kPreviousTotalVoteSWOP = "previous_total_vote_SWOP"
let keySwopYearEmission = "swop_year_emission"
let keyBalancecpmmA = "A_asset_balance"
let keyBalancecpmmB = "B_asset_balance"
let kHarvestPoolActiveVoteStrucVoting = "harvest_pool_activeVote_struc"
let kHarvestUserPoolActiveVoteStrucVoting = "_harvest_user_pool_activeVote_struc"
let keyLimitShareFirstHarvest = "share_limit_on_first_harvest"
let keyAssetIdA = "A_asset_id"
let keyAssetIdB = "B_asset_id"
let keyFirstHarvestHeight = "first_harvest_height"
let keyfirstHarvestCpmm = "first_harvest"
let keyTempPrevSum = "sum_reward_previous"
let keyTempCurSum = "sum_reward_current"
let governanceAddress = Address(base58'3PLHVWCqA9DJPDbadUofTohnCULLauiDWhS')
let wallet = Address(base58'3P6J84oH51DzY6xk2mT5TheXRbrCwBMxonp')
let votingAddress = Address(base58'3PQZWxShKGRgBN1qoJw6B4s9YWS9FneZTPg')
let adminIncreaseInterestAddress = Address(base58'3PPupsBVHgDXaRhyMbkTxminzAotp8AMsr6')
let oneWeekInBlock = 10106 # 1440760/59.84 where 59.84 - real block duration in seconds
let totalVoteShare = 10000000000
let scaleValue1 = 10
let scaleValue3 = 1000
let scaleValue5 = 100000
let scaleValue6 = 1000000
let scaleValue8 = 100000000
let scaleValue11 = 100000000000
```

```
func strAssetIdA(pool:Address)= pool.getStringValue(keyAssetIdA)
func strAssetIdB(pool:Address) = pool.getStringValue(keyAssetIdB)
func assetIdA(pool:Address) = if strAssetIdA(pool) == "WAVES" then unit else
strAssetIdA(pool).fromBase58String()
func assetIdB(pool:Address) = if strAssetIdB(pool) == "WAVES" then unit else
strAssetIdB(pool).fromBase58String()
let kBasePeriod = "base_period"
let kPeriodLength = "period_length"
let kStartHeight = "start_height"
let kFirstHarvestHeight = "first_harvest_height"
let kDurationFullVotePower = "duration_full_vote_power"
let kMinVotePower = "min_vote_power"

let basePeriod = votingAddress.getInteger(kBasePeriod).valueOrErrorMessage("Empty kBasePeriod") # 0
let startHeight = votingAddress.getInteger(kStartHeight).valueOrErrorMessage("Empty kStartHeight")
let periodLength = votingAddress.getInteger(kPeriodLength).valueOrErrorMessage("Empty kPeriodLength")
# 10102
let durationFullVotePower =
votingAddress.getInteger(kDurationFullVotePower).valueOrErrorMessage("Empty kDurationFullVotePower")
# 1443, 1 days in block after voting start
let minVotePower = votingAddress.getInteger(kMinVotePower).valueOrErrorMessage("Empty
kMinVotePower") # 10000000, minVoteCoeff/scale8 between 0 and 1 - voting power if vote at the end of
voting period

let isActive = this.getBooleanValue(keyActive)

let currPeriod = basePeriod + (height-startHeight)/periodLength
func getLimitToken(pool:Address) = pool.getIntegerValue(keyLimitShareFirstHarvest).valueOrElse(0)

let APY = this.getIntegerValue(keyAPY)
let SwopYearEmission = this.getIntegerValue(keySwopYearEmission)

func assetNameA(pool:Address) = match assetIdA(pool) {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}
func assetNameB(pool:Address) = match assetIdB(pool) {
case id: ByteVector => assetInfo(id).value().name
case waves: Unit => "WAVES"
}

let SWOP = this.getStringValue(keySWOPid).fromBase58String()
func isFirstHarvest(pool:Address) = pool.getBoolean(keyfirstHarvestCpmm).valueOrElse(false)

func getHeightFirstHarvest(pool:Address) = pool.getInteger(keyFirstHarvestHeight).valueOrElse(0)

func getBalanceA(pool:Address) = pool.getInteger(keyBalancecpmmA)
.valueOrErrorMessage("No data on the key: " + keyBalancecpmmA)

func getBalanceB(pool:Address) = pool.getInteger(keyBalancecpmmB)
.valueOrErrorMessage("No data on the key: " + keyBalancecpmmB)
```

```
func getShareLimitToken(pool:Address) = pool.getInteger(kShareLimit)
.valueOrErrorMessage("No data on the key: " + kShareLimit)

func getTotalShareTokenLocked(pool:String) = this.getInteger(pool + keyShareTokensLocked)
.valueOrErrorMessage("No data on the key: " + pool + keyShareTokensLocked)

func getShareAssetId(pool:String) =
pool.addressFromString().value().getStringValue("share_asset_id").fromBase58String()

func accountBalance(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector => this.assetBalance(id)
case waves: Unit => this.wavesBalance().available
}

func getAssetInfo(assetId: ByteVector|Unit) = match assetId {
case id: ByteVector =>
let stringId = id.toBase58String()
let info = assetInfo(id).valueOrErrorMessage("Asset " + stringId + " doesn't exist")
(stringId, info.name, info.decimals)
case waves: Unit => ("WAVES", "WAVES", 8)
}
```

- Checkpoint1: Calculate scale value for assetId1/assetId2 with 8 digits after delimiter

```
func calcScaleValue(assetId1:ByteVector,assetId2:ByteVector) = {
let assetId1Decimals = assetId1.assetInfo().value().decimals
let assetId2Decimals = assetId2.assetInfo().value().decimals
let scaleDigits = assetId2Decimals-assetId1Decimals+8
pow(10,0,scaleDigits,0,0,HALFDOWN)
}

func userAvailableSWOP(pool:String, user:Address) = this.getInteger(pool+"_"+user.toString() +
keyAvailableSWOP).valueOrElse(0)

func rewardInfo(pool:String) = {
let totalRewardPerBlockCurrent = governanceAddress.getInteger(keyTotalRewardPerBlockCurrent)
.valueOrErrorMessage("No data on the key: " + keyTotalRewardPerBlockCurrent + " at address " +
governanceAddress.toString())
let totalRewardPerBlockPrevious = governanceAddress.getInteger(keyTotalRewardPerBlockPrevious)
.valueOrErrorMessage("No data on the key: " + keyTotalRewardPerBlockPrevious + " at address " +
governanceAddress.toString())
let rewardPoolFractionCurrent = governanceAddress.getInteger(pool + keyRewardPoolFractionCurrent)
.valueOrErrorMessage("No data on the key: " + pool + keyRewardPoolFractionCurrent + " at address " +
governanceAddress.toString())
let rewardUpdateHeight = governanceAddress.getInteger(keyRewardUpdateHeight)
.valueOrErrorMessage("No data on the key: " + keyRewardUpdateHeight + " at address " +
governanceAddress.toString())
let poolRewardUpdateHeight = governanceAddress.getInteger(pool + keyHeightPoolFraction)
.valueOrElse(0)
```

```
let rewardPoolFractionPrevious = governanceAddress.getInteger(pool + keyRewardPoolFractionPrevious)
.valueOrErrorMessage("No data on the key: " + pool + keyRewardPoolFractionPrevious + " at address " +
governanceAddress.toString())

let rewardPoolCurrent = totalRewardPerBlockCurrent.fraction(rewardPoolFractionCurrent,totalVoteShare)
let rewardPoolPrevious =
totalRewardPerBlockPrevious.fraction(rewardPoolFractionPrevious,totalVoteShare)

if rewardPoolCurrent > totalRewardPerBlockCurrent || rewardPoolPrevious > totalRewardPerBlockPrevious
then
throw("rewardPoolCurrent > totalRewardPerBlockCurrent or rewardPoolPrevious >
totalRewardPerBlockPrevious")
else
(rewardPoolCurrent,rewardUpdateHeight,rewardPoolPrevious, poolRewardUpdateHeight)
}
```

- Checkpoint2: Get Last Interest

```
func getLastInterestInfo(pool:String) = {
let lastInterest = this.getInteger(pool + keyLastInterest)
.valueOrErrorMessage("No data on the key: " + pool + keyLastInterest)
let lastInterestHeight = this.getInteger(pool + keyLastInterestHeight)
.valueOrElse(height)
(lastInterestHeight,lastInterest)
}
```

- Checkpoint3: Get User Interest

```
func getUserInterestInfo(pool:String, userAddress:Address) = {
let userLastInterest = this.getInteger(pool + "" + userAddress.toString() + keyUserLastInterest)
let userShare = this.getInteger(pool + "" + userAddress.toString() + keyUserShareTokensLocked)
let lastInterest = this.getInteger(pool + keyLastInterest)
.valueOrErrorMessage("No data on the key: " + pool + keyLastInterest)
let userLastInterestValue = match userLastInterest {
case userLastInterest: Int => userLastInterest
case _ => lastInterest
}
let userShareTokensAmount = match userShare {
case userShare: Int => userShare
case _ => 0
}
(userLastInterestValue,userShareTokensAmount)
}
```

- Checkpoint4: Get Reward Based on Block Height

func
calcInterest(lastInterestHeight:Int,rewardUpdateHeight:Int,poolRewardUpdateHeight:Int,lastInterest:Int,currentRewardPerBlock:Int,shareTokenLocked:Int,previousRewardPerBlock:Int,shareAssetId:ByteVector,scaleValue:Int,pmtAmount:Int) = {
if shareTokenLocked == 0 then 0 else # this condition true only for the first call

if (poolRewardUpdateHeight != 0) then {
if height < rewardUpdateHeight && rewardUpdateHeight == poolRewardUpdateHeight then # in case of updating reward info at gov dApp
let reward = previousRewardPerBlock*(height-lastInterestHeight)*
lastInterest + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
else if rewardUpdateHeight < height && rewardUpdateHeight != poolRewardUpdateHeight then
let reward = previousRewardPerBlock(height-lastInterestHeight)
lastInterest + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
else if rewardUpdateHeight < height && rewardUpdateHeight == poolRewardUpdateHeight && lastInterestHeight > rewardUpdateHeight then
let reward = currentRewardPerBlock*(height-lastInterestHeight)*
lastInterest + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
else # in case if reward updated we need update interest
let rewardAfterLastInterestBeforeReawardUpdate = previousRewardPerBlock(rewardUpdateHeight-lastInterestHeight)
let interestAfterUpdate = lastInterest + fraction(rewardAfterLastInterestBeforeReawardUpdate,scaleValue,shareTokenLocked)
let reward = currentRewardPerBlock*(height-rewardUpdateHeight)*
interestAfterUpdate + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
} else {
if height < rewardUpdateHeight then # in case of updating reward info at gov dApp
let reward = previousRewardPerBlock(height-lastInterestHeight)
lastInterest + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
else
if (lastInterestHeight > rewardUpdateHeight) then
let reward = currentRewardPerBlock*(height-lastInterestHeight)*
lastInterest + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
else # in case if reward updated we need update interest
let rewardAfterLastInterestBeforeReawardUpdate = previousRewardPerBlock(rewardUpdateHeight-lastInterestHeight)
let interestAfterUpdate = lastInterest + fraction(rewardAfterLastInterestBeforeReawardUpdate,scaleValue,shareTokenLocked)
let reward = currentRewardPerBlock*(height-rewardUpdateHeight)

interestAfterUpdate + fraction(reward,scaleValue,shareTokenLocked) # interests[n] = interests[n-1] + reward[n] / totalSupplyLP(n)
}
}

func claimCalc(pool:String,caller:Address,pmtAmount:Int) = {
let shareAssetId = getShareAssetId(pool)
let scaleValue = calcScaleValue(SWOP,shareAssetId) # return scale8 value with considering digits in assets
let shareTokenLocked = getTotalShareTokenLocked(pool)
let (lastInterestHeight,lastInterest) = getLastInterestInfo(pool)
let (currentRewardPerBlock,rewardUpdateHeight,previousRewardPerBlock, poolRewardUpdateHeight) = rewardInfo(pool)
let (userLastInterest,userShareTokensAmount) = getUserInterestInfo(pool,caller)
let currentInterest = calcInterest(lastInterestHeight,rewardUpdateHeight,poolRewardUpdateHeight,lastInterest,currentRewardPerBlock,shareTokenLocked,previousRewardPerBlock,shareAssetId,scaleValue,pmtAmount) # multiple by scale8
let claimAmount = fraction(userShareTokensAmount,currentInterest-userLastInterest,scaleValue)
let userNewInterest = currentInterest
(userNewInterest,currentInterest,claimAmount,userShareTokensAmount)
}

- Checkpoint5: added update pools

func calculateProtocolReward(pool:String) ={
let (lastInterestHeight,lastInterest) = getLastInterestInfo(pool)
let (currentRewardPerBlock,rewardUpdateHeight,previousRewardPerBlock,poolRewardUpdateHeight) = rewardInfo(pool)
let shareTokenLocked = getTotalShareTokenLocked(pool)
if shareTokenLocked == 0 && poolRewardUpdateHeight == 0 then
if height < rewardUpdateHeight then
let reward = previousRewardPerBlock*(height-lastInterestHeight)*
reward
else
if (lastInterestHeight > rewardUpdateHeight) then
let reward = currentRewardPerBlock(height-lastInterestHeight)
reward
else
let rewardAfterLastInterestBeforeReawardUpdate = previousRewardPerBlock*(rewardUpdateHeight-lastInterestHeight)*
let reward = currentRewardPerBlock(height-rewardUpdateHeight)
reward + rewardAfterLastInterestBeforeReawardUpdate
else if shareTokenLocked == 0 && poolRewardUpdateHeight != 0 then
if height < rewardUpdateHeight && rewardUpdateHeight == poolRewardUpdateHeight then # in case of updating reward info at gov dApp
let reward = previousRewardPerBlock*(height-lastInterestHeight)*
reward
else if rewardUpdateHeight < height && rewardUpdateHeight != poolRewardUpdateHeight then

```
let reward = previousRewardPerBlock(height-lastInterestHeight)
reward
else if rewardUpdateHeight < height && rewardUpdateHeight == poolRewardUpdateHeight &&
lastInterestHeight > rewardUpdateHeight then
let reward = currentRewardPerBlock(height-lastInterestHeight)
reward
else # in case if reward updated we need update interest
let rewardAfterLastInterestBeforeReawardUpdate = previousRewardPerBlock(rewardUpdateHeight-
lastInterestHeight)
let reward = currentRewardPerBlock*(height-rewardUpdateHeight)
reward + rewardAfterLastInterestBeforeReawardUpdate
else
0
}

func checkPmtAssetIdCorrect(pool:String,pmtAssetId:ByteVector|Unit) = {
let poolShareAssetId =
pool.addressFromString().value().getStringValue("share_asset_id").fromBase58String()
if pmtAssetId == poolShareAssetId then true else false
}

func getUserSWOPClaimedAmount(pool:String,user:Address) = this.getInteger(pool + "_" + user.toString() +
keyUserSWOPClaimedAmount)
.valueOrElse(0)

func suspend(cause: String) = [
BooleanEntry(keyActive, false),
StringEntry(keyCause, cause)
]

@Callable(i)
func init(earlyLP:String) = {
if this.getString(keySWOPid).isDefined() then throw("SWOP already initialized") else
let initAmount = 100000000000000
let SWOPissue = Issue("SWOP", "SWOP protocol token", initAmount, 8, true)
let SWOPid = SWOPissue.calculateAssetId()
[
BooleanEntry(keyActive, true),
Issue("SWOP", "SWOP protocol token", initAmount, 8, true),
StringEntry(keySWOPid, SWOPid.toBase58String())
]
}

@Callable(i)
func initPoolShareFarming(pool:String) = {
if i.caller != this then
throw("Only the DApp itself can call this function") else
let (currentReward,rewardUpdateHeight,previousRewardPerBlock,poolRewardUpdateHeight) =
rewardInfo(pool)
```

```
[
IntegerEntry(pool + keyShareTokensLocked, 0),
IntegerEntry(pool + keyLastInterest, 0),
IntegerEntry(pool + keyLastInterestHeight,height)
]
}

@Callable(i)
func updatePoolInterest(pool:String) = {
if i.caller != wallet then
throw("Only the Admin itself can call this function")
else if !isActive then
throw("DApp is inactive at this moment") else
let (userNewInterest,currentInterest,claimAmount,userShareTokensAmount) =
claimCalc(pool,adminIncreaseInterestAddress,0)
let (currentReward,rewardUpdateHeight,previousRewardPerBlock,poolRewardUpdateHeight) =
rewardInfo(pool)
[
IntegerEntry(pool + keyLastInterest, userNewInterest),
IntegerEntry(pool + keyLastInterestHeight,height)
]
}

@Callable(i)
func lockShareTokens(pool:String) = {
let (pmtAmount, pmtAssetId) = (i.payments[0].amount, i.payments[0].assetId)
let (pmtStrAssetId, pmtAssetName, pmtDecimals) = getAssetInfo(pmtAssetId)
let (userNewInterest,currentInterest,claimAmount,userShareTokensAmount) =
claimCalc(pool,i.caller,pmtAmount)
let userShareAmountNew = userShareTokensAmount+pmtAmount
let availableFundsNew = userAvailableSWOP(pool,i.caller) + claimAmount
let totalShareAmount = getTotalShareTokenLocked(pool)
let totalShareAmountNew = totalShareAmount + pmtAmount
let userClaimedAmount = getUserSWOPClaimedAmount(pool,i.caller)
let userClaimedAmountNew = userClaimedAmount + claimAmount
let baseEntry = [
IntegerEntry(pool + "" + i.caller.toString() + keyUserLastInterest, userNewInterest),
IntegerEntry(pool + "" + i.caller.toString() + keyUserShareTokensLocked, userShareAmountNew),
IntegerEntry(pool + keyShareTokensLocked, totalShareAmountNew),
IntegerEntry(pool + keyLastInterest, currentInterest),
IntegerEntry(pool + keyLastInterestHeight, height),
IntegerEntry(pool + "" + i.caller.toString() + keyUserSWOPClaimedAmount, userClaimedAmountNew),
IntegerEntry(pool + "" + i.caller.toString() + keyUserSWOPLastClaimedAmount, claimAmount),
IntegerEntry(pool + "_" + i.caller.toString() + keyAvailableSWOP, availableFundsNew)
]
```

```
if (pmtAmount <= 0) then
throw("You can't lock token")
else if !isActive then
throw("DApp is inactive at this moment")
else if !checkPmtAssetIdCorrect(pool,pmtAssetId) then throw("Incorrect pmtAssetId")
else if (isFirstHarvest(Address(pool.fromBase58String())) &&
getHeightFirstHarvest(Address(pool.fromBase58String())) > height) then {
let harvestPeriod = (getHeightFirstHarvest(Address(pool.fromBase58String())) - startHeight + 1) /
periodLength - 1
let amountOfVoting = votingAddress.getStringValue(i.caller.toString() + "" + pool + "user_pool_struc").split("")
let amountPoolStract = votingAddress.getStringValue(pool + "pool_struc").split("")
let amountActiveVoteUserPoolStract = votingAddress.getString(i.caller.toString() + "" + pool +
kHarvestUserPoolActiveVoteStrucVoting).valueOrElse("").split("")
let amountPoolActiveVoteStract = votingAddress.getString(pool +
kHarvestPoolActiveVoteStrucVoting).valueOrElse("").split("")
let userShareTokenLocked = userShareTokensAmount
let userPoolActiveVote = if currPeriod.toString() == amountOfVoting[2] then
{parseInt(amountActiveVoteUserPoolStract[0]).valueOrElse(0)} else
{parseInt(amountOfVoting[1]).valueOrElse(0)}
let poolActiveVote = if currPeriod.toString() == amountPoolStract[2] then
{parseInt(amountPoolActiveVoteStract[0]).valueOrElse(0)} else
{parseInt(amountPoolStract[1]).valueOrElse(0)}
let protocolReward = calculateProtocolReward(pool)
if(userPoolActiveVote != 0) then {
let limitShareToken = getShareLimitToken(pool.addressFromStringValue())
let shareToken = fraction(limitShareToken,userPoolActiveVote,poolActiveVote) - userShareTokenLocked
if amountActiveVoteUserPoolStract.size() > 1 &&
parseInt(amountActiveVoteUserPoolStract[1]).valueOrElse(0) >= harvestPeriod then
throw("You can't share token")
else if (pmtAmount > limitShareToken) then
throw("You can't share token more than " + limitShareToken.toString())
else if (shareToken > 0) then {
if totalShareAmountNew < fraction(99, (accountBalance(pmtAssetId) + pmtAmount), 100) then
throw("Balance of share-token is greater than totalAmount")
else if (totalShareAmount == 0) then {
baseEntry ++ [
Reissue(SWOP, protocolReward, true),
ScriptTransfer(wallet, protocolReward, SWOP) ]
} else if (pmtAmount <= shareToken) then {
baseEntry
} else
throw("Your maximum share token is " + shareToken.toString())
} else
throw("You can't share token")
} else {
throw("Your amount of token less than 0")
}
```

```
}
else {
baseEntry
}
}

@Callable(i)
func withdrawShareTokens(pool:String,shareTokensWithdrawAmount:Int) = {
let shareTokensId = pool.addressFromString().value().getStringValue("share_asset_id").fromBase58String()
let (userNewInterest,currentInterest,claimAmount,userShareTokensAmount) = claimCalc(pool,i.caller,1)
let userShareAmountNew = userShareTokensAmount-shareTokensWithdrawAmount
let availableFundsNew = userAvailableSWOP(pool,i.caller) + claimAmount
let totalShareAmount = getTotalShareTokenLocked(pool)
let totalShareAmountNew = totalShareAmount - shareTokensWithdrawAmount
let userClaimedAmount = getUserSWOPClaimedAmount(pool,i.caller)
let userClaimedAmountNew = userClaimedAmount + claimAmount

if shareTokensWithdrawAmount > userShareTokensAmount then
throw("Withdraw amount more then user locked amount")
else if !isActive then
throw("DApp is inactive at this moment") else
if shareTokensWithdrawAmount > userShareTokensAmount then throw("Withdraw amount more then user
locked amount")
else if totalShareAmountNew < fraction(99, (accountBalance(shareTokensId) -
shareTokensWithdrawAmount), 100) then throw("Balance of share-token is greater than totalAmount") else
[
IntegerEntry(pool + "" + i.caller.toString() + keyUserLastInterest, userNewInterest),
IntegerEntry(pool + "" + i.caller.toString() + keyUserShareTokensLocked, userShareAmountNew),
IntegerEntry(pool + keyLastInterest, currentInterest),
IntegerEntry(pool + keyLastInterestHeight, height),
IntegerEntry(pool + keyShareTokensLocked, totalShareAmountNew),
IntegerEntry(pool+""+i.caller.toString() + keyAvailableSWOP, availableFundsNew),
IntegerEntry(pool + "" + i.caller.toString() + keyUserSWOPClaimedAmount, userClaimedAmountNew),
IntegerEntry(pool + "_" + i.caller.toString() + keyUserSWOPLastClaimedAmount, claimAmount),
ScriptTransfer(i.caller, shareTokensWithdrawAmount, shareTokensId)
]
}
```

- Checkpoint6: Claim Tokens from Pools

```
@Callable(i)
func claim(pool:String) = {
let shareTokensId = pool.addressFromString().value().getStringValue("share_asset_id").fromBase58String()
let shareTokenLocked = getTotalShareTokenLocked(pool)
let (lastInterestHeight,lastInterest) = getLastInterestInfo(pool)
let (currentRewardPerBlock, rewardUpdateHeight,previousRewardPerBlock,poolRewardUpdateHeight) =
rewardInfo(pool)
let (userNewInterest,currentInterest,claimAmount,userShareTokensAmount) = claimCalc(pool,i.caller,1)
```

let availableFund = userAvailableSWOP(pool,i.caller) + claimAmount
let userClaimedAmount = getUserSWOPClaimedAmount(pool,i.caller)
let userClaimedAmountNew = userClaimedAmount + claimAmount

- Checkpoint7: Check states before claim

if availableFund == 0 then
throw("You have 0 available SWOP")
else if !isActive then
throw("DApp is inactive at this moment") else
if availableFund == 0 then throw("You have 0 available SWOP")
else if shareTokenLocked < fraction(99, accountBalance(shareTokensId), 100) then throw("Balance of share-token is greater than totalAmount") else
[
IntegerEntry(pool + "" + *i.caller.toString() + keyUserLastInterest, userNewInterest),*
IntegerEntry(pool + keyLastInterest, currentInterest),
IntegerEntry(pool + keyLastInterestHeight, height),
IntegerEntry(pool + "" + i.caller.toString() + keyAvailableSWOP, 0),
Reissue(SWOP, availableFund, true),
IntegerEntry(pool + "" + *i.caller.toString() + keyUserSWOPClaimedAmount, userClaimedAmountNew),*
IntegerEntry(pool + "" + i.caller.toString() + keyUserSWOPLastClaimedAmount, claimAmount),

- Checkpoint8: Transfer

ScriptTransfer(i.caller, availableFund, SWOP)
]
}

@Callable(i)
func shutdown() =
if !isActive then
throw("DApp is already suspended. Cause: " + this.getString(keyCause).valueOrElse("the cause wasn't specified"))
else if ![adminPubKey1, adminPubKey2, adminPubKey3].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else
suspend("Paused by admin")
@Callable(i)
func activate() =
if isActive then
throw("DApp is already active")
else if ![adminPubKey1, adminPubKey2, adminPubKey3].containsElement(i.callerPublicKey) then
throw("Only admin can call this function")
else [
BooleanEntry(keyActive, true),
DeleteEntry(keyCause)
]

```
@Verifier(tx)
func verify() = match tx {
case _ => {
let adminPubKey1Signed = if sigVerify(tx.bodyBytes, tx.proofs[0], adminPubKey1) then 1 else 0
let adminPubKey2Signed = if sigVerify(tx.bodyBytes, tx.proofs[1], adminPubKey2) then 1 else 0
let adminPubKey3Signed = if sigVerify(tx.bodyBytes, tx.proofs[2], adminPubKey3) then 1 else 0
adminPubKey1Signed + adminPubKey2Signed + adminPubKey3Signed >= 2
}
}
```

10. List of issues by severity

A. Critical

- N/A

B. High

- N/A

C. Medium

- N/A

D. Low

- N/A

11. List of issues by source file

- N/A

12. Issue descriptions

- N/A

13. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- N/A