

Exoswap Whitepaper

JJ Uzumaki
jjuzumaki@proton.me
www.exohood.org

MAY 2022

Introduction

Exoswap is a noncustodial automated protocol which generates value for the decentralised finance ecosystem through a Binance Smart Chain, which can be implemented in any decentralised project for a unique adoption environment. Easy to use and efficient gas rate which generates greater security for users. This Widget/API will be helpful for businesses, projects and will work particularly well as a component of other smart contracts that require guaranteed on-chain liquidity. Most exchanges maintain an order book and facilitate matching between buyers and sellers. Exoswap smart contracts hold liquidity reserves of various tokens, and transactions are executed directly against these reserves. Prices are set automatically using the constant product market-making mechanism ($e \cdot x = p$), keeping general reserves in relative balance. Reserves are pooled among a network of liquidity providers who supply tokens to the system in exchange for a proportional share of transaction fees.

An essential feature of Exoswap is using a dex/labs contract that implements a separate exchange contract for each BEP20 token. Each of these exchange contracts has a reserve of BNB and its associated BEP20.

This allows trades between the two based on relative supply. Swap contracts are linked through the registry, allowing direct BEP20 to BEP20 swaps between any token using BNB as an intermediary. The implementation will be easy to use for anyone without programming knowledge, once the smart contract is launched, a code is generated to be copied and pasted on the website, where the exchange will work as a widget interface.

1. Building Decentralised Value

<Exoswap_lab.vy> is a smart contract used to manufacture and register Exoswap trading contracts. The createDex() public function allows any Binance Smart Chain user to implement a trade contract for any BEP20 that doesn't already have one.

Example:

```
DexTemplate: public(address)
token_to_Dex: address[address]
Dex_to_token: address[address]
```

```
@public
def __init__(template: address):
    self.DexTemplate = template
```

```
@public
def createDex(token: address) -> address:
    assert self.token_to_Dex[token] == ZERO_ADDRESS
    new_Dex: address = create_with_code_of(self.DexTemplate)
```

```

self.token_to_Dex[token] = new_Dex
self.Dex_to_token[new_Dex] = token
return new_Dex

```

The Lab stores a record of all tokens and their associated exchanges. With a token or exchange address, the `getDex()` and `getToken()` functions can be used to look up the other.

Example:

```

@public
@constant
def getDex(token: address) -> address:
    return self.token_to_Dex[token]

```

```

@public
@constant
def getToken(Dex: address) -> address:
    return self.Dex_to_token[Dex]

```

The lab doesn't perform any verification on a token when a trade contract is launched, only a limit of one trade per token is met. Users and interfaces should only interact with exchanges associated with tokens they trust.

2. BNB \rightleftharpoons BEP20 Trades

Each exchange contract (`exoswap_dex.vy`) is associated with a single BEP20 token and has a liquidity pool of BNB and that token. The exchange rate between BNB and a BEP20 is based on the relative sizes of their liquidity pools within the contract. This is done by keeping the relationship $\text{bnb_pool} * \text{token_pool} = \text{unchanged}$. This invariant remains constant during trades and only changes when liquidity is added or removed from the market.

Below is a simplified version of `bnbToTokenSwap()`, the function for converting BNB to BEP20 tokens:

```

bnb_pool: uint256
token_pool: uint256
token: address(BEP20)

@public
@payable
def ethToTokenSwap():
    fee: uint256 = msg.value / 500
    invariant: uint256 = self.eth_pool * self.token_pool
    new_eth_pool: uint256 = self.eth_pool + msg.value
    new_token_pool: uint256 = invariant / (new_eth_pool - fee)
    tokens_out: uint256 = self.token_pool - new_token_pool
    self.eth_pool = new_eth_pool
    self.token_pool = new_token_pool
    self.token.transfer(msg.sender, tokens_out)

```

Note: For gas efficiency, bnb_pool and token_pool are not stored variables. They are found using self.balance and through an external call to self.token.balanceOf(self)

When BNB is sent to the function, bnb_pool increases. To keep the relationship $\text{bnb_pool} * \text{token_pool} = \text{unchanged}$, the pool of tokens is reduced by a proportional amount. The amount by which the token_pool is reduced is the number of tokens purchased. This change in the reserve index turns the BNB exchange rate to BEP20, incentivising trades in the opposite direction.

The exchange of tokens for BNB is done with the tokenToBnbSwap() function:

```
@public
def tokenToBnbSwap(tokens_in: uint256):
    fee: uint256 = tokens_in / 500
    invariant: uint256 = self.bnb_pool * self.token_pool
    new_token_pool: uint256 = self.token_pool + tokens_in
    new_bnb_pool: uint256 = self.invariant / (new_token_pool - fee)
    bnb_out: uint256 = self.bnb_pool - new_bnb_pool
    self.bnb_pool = new_bnb_pool
    self.token_pool = new_token_pool
    self.token.transferFrom(msg.sender, self, tokens_out)
    send(msg.sender, bnb_out)
```

This increases token_pool and decreases bnb_pool, changing the price in the opposite direction. Below is an example of buying BNB from EXO.

Example: BNB → EXO

Note: This example uses a rate of 0.4% as it is an example implementation. The Exoswap contract has a commission of 0.3%.

The liquidity providers deposit 1 BNB and 15,000 EXO (BEP20) into a smart contract. An invariant is automatically set, which makes $\text{BNB_pool} * \text{EXO_pool} = \text{invariant}$.

Example:

BNB_pool = 2

EXO_pool = 15000

invariant = 2 * 15000 = 30000

An EXO buyer sends 1 BNB to the contract. Liquidity providers charge a 0.4% fee and the remaining 0.996 BNB is added to the BNB_pool. The invariant is then divided by the new amount of BNB in the liquidity pool to determine the new EXO_pool size. The remaining EXO is sent to the buyer.

Buyer sends: 1 BNB

Fee = 1 BNB / 15000 = 0,00006667 BNB

BNB_pool = 2 + 1 - 0,00006667 = 2,99993333

EXO_pool = 30000/2,99993333 = 10000,2222

Buyer receives: 15000 - 10000,2222 = 4999,7778 EXO

The fee is now added back to the liquidity pool, which acts as a payment to liquidity providers that is collected when liquidity is withdrawn from the market. Since the fee is added after the price calculation, the invariant increases slightly with each trade, making the system profitable for liquidity providers. In fact, what the invariant actually represents is BNB_pool * EXO_pool at the end of the above transaction.

BNB_pool = 2,99993333 + 0,00006667 = 3

EXO_pool = 10000,2222

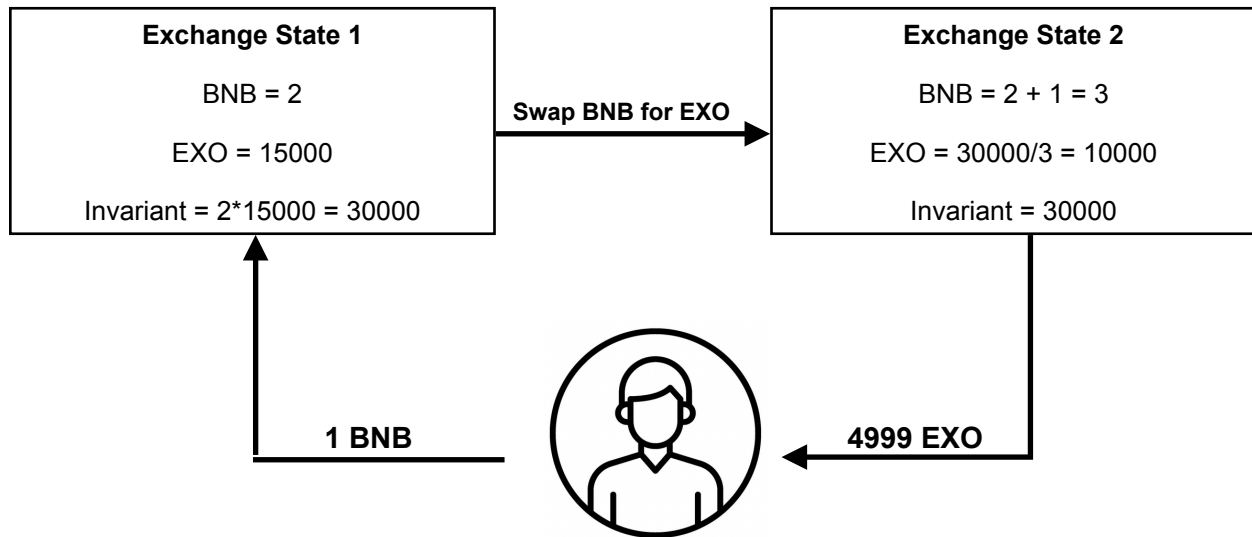
new invariant = 3 * 10000,2222 = 30000,6666

In this case, the buyer received a rate of 4999.7778 EXO/BNB. However, the price has changed. If another buyer makes a trade in the same direction, he will get a slightly lower EXO/BNB rate. However, if a buyer trades in the opposite direction, he will get a higher BNB/EXO rate.

1 BNB in
4995 EXO out
Rate = 4999,7778 EXO/BNB

Purchases that are largely relative to the total size of the liquidity pools will cause a price drop. In an active market, arbitrage will ensure that the price doesn't stray too far from other exchanges.

BNB to EXO Exchange in Exoswap



3. BEP20 \rightleftharpoons BEP20 Trades

Since BNB is used as a common pair for all BEP20 tokens, it can be used as an intermediary for direct BEP20 to BEP20 trades. For example, it's possible to convert EXO to BNB on one exchange and then BNB to MUS on another within a single transaction.

To convert EXO to MUS (for example), a buyer calls the `tokenToTokenSwap()` function on the EXO swap contract:

```
contract lab():  
    def getExchange(token_addr: address) -> address: constant
```

```
contract Exchange():  
    def bnbToTokenTransfer(recipient: address) -> bool: modifying
```

```
lab: Lab
```

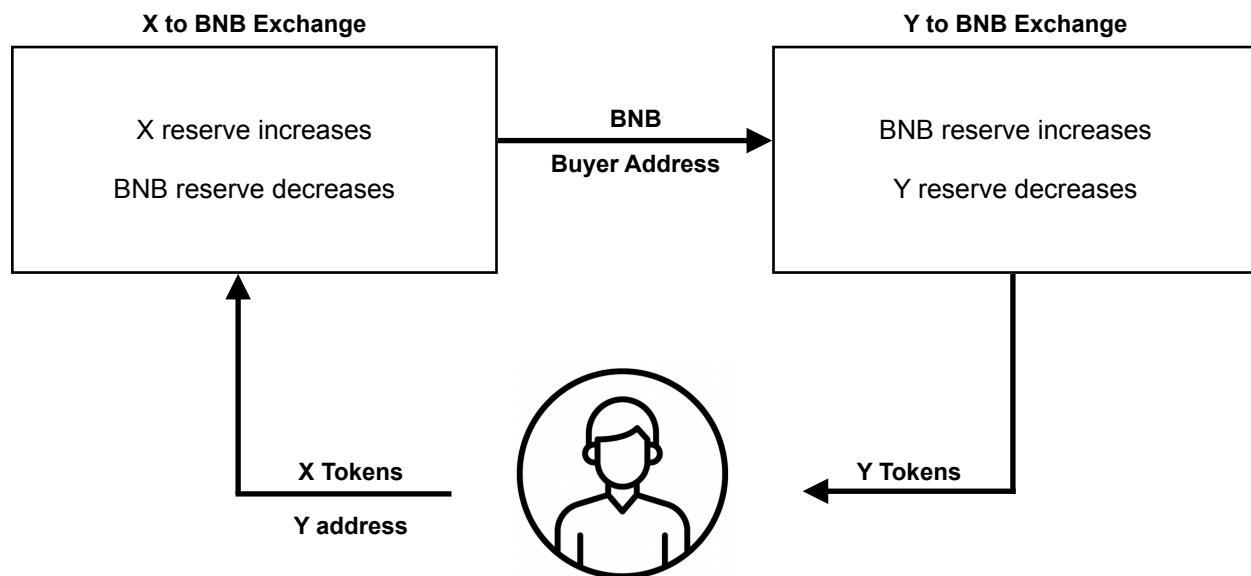
```
@public
```

```
def tokenToTokenSwap(token_addr: address, tokens_sold: uint256):  
    exchange: address = self.lab.getExchange(token_addr)  
    fee: uint256 = tokens_sold / 500  
    invariant: uint256 = self.bnb_pool * self.token_pool  
    new_token_pool: uint256 = self.token_pool + tokens_sold  
    new_bnb_pool: uint256 = invariant / (new_token_pool - fee)  
    bnb_out: uint256 = self.bnb_pool - new_bnb_pool  
    self.bnb_pool = new_bnb_pool  
    self.token_pool = new_token_pool  
    Exchange(exchange).bnbToTokenTransfer(msg.sender, value=bnb_out)
```

where <token_addr> is the address of the MUS token and tokens_sold is the amount of EXO being sold. This function first checks the factory to retrieve the MUS exchange address. The exchange then converts the EXO entry to BNB. However, instead of returning the purchased BNB to the buyer, the function calls the bnbToTokenTransfer() payment function on the MUS exchange:

```
@public
@payable
def bnbToTokenTransfer(recipient: address):
    fee: uint256 = msg.value / 500
    invariant: uint256 = self.bnb_pool * self.token_pool
    new_bnb_pool: uint256 = self.bnb_pool + msg.value
    new_token_pool: uint256 = invariant / (new_bnb_pool - fee)
    tokens_out: uint256 = self.token_pool - new_token_pool
    self.bnb_pool = new_bnb_pool
    self.token_pool = new_token_pool
    self.invariant = new_bnb_pool * new_token_pool
    self.token.transfer(recipient, tokens_out)
```

bnbToTokenTransfer() receives the BNB and address from the buyer, verifies that the call is from an exchange on the registry, converts the BNB to MUS, and forwards the MUS to the original buyer. bnbToTokenTransfer() functions identically to bnbToTokenSwap() but has the additional input parameter recipient: address. This is used to forward purchased tokens to the original buyer instead of msg.sender, which in this case would be the Exoswap.



4. Swaps vs Transfers

The `bnbToTokenSwap()`, `tokenToBnbSwap()` and `tokenToTokenSwap()` functions return the purchased tokens to the buyers address.

The `bnbToTokenTransfer()`, `tokenToBnbTransfer()`, and `tokenToTokenTransfer()` functions allow buyers to make a trade and then immediately transfer the purchased tokens to a recipient's address.

5. Providing Liquidity

5.1 Adding Liquidity

Adding liquidity requires depositing an equivalent value of BNB and BEP20 tokens into the associated BEP20 token exchange contract.

The first liquidity provider to join a pool sets the initial exchange rate by depositing what it thinks the equivalent value of BNB and BEP20 tokens are. If this ratio is off, arbitrage traders will balance prices at the expense of the initial liquidity provider.

All future liquidity providers deposit BNB and BEP20 using the exchange rate at the time of your deposit. If the exchange rate is low, there is a profitable arbitrage opportunity that will correct the price.

Liquidity Tokens

Liquidity tokens are minted to track the relative proportion of total reserves that each liquidity provider has contributed. They are highly divisible and can be burned at any time to return a proportionate share of the markets' liquidity to the provider.

This is what Liquidity providers call the `addLiquidity()` function to deposit into reserves and mint new liquidity tokens:

```
@public
@payable
def addLiquidity():
    token_amount: uint256 = msg.value * token_pool / bnb_pool
    liquidity_minted: uint256 = msg.value * total_liquidity / bnb_pool

    bnb_added: uint256 = msg.value
    shares_minted: uint256 = (bnb_added * self.total_shares) / self.bnb_pool
    tokens_added: uint256 = (shares_minted * self.token_pool) / self.total_shares
    self.shares[msg.sender] = self.shares[msg.sender] + shares_minted
    self.total_shares = self.total_shares + shares_minted
    self.bnb_pool = self.bnb_pool + bnb_added
    self.token_pool = self.token_pool + tokens_added
    self.token.transferFrom(msg.sender, self, tokens_added)
```

The amount of liquidity tokens minted is determined by the amount of BNB sent to the feature. It can be calculated using the equation:

$$amountMinted = totalAmount * \frac{bnbDeposited}{bnbPool}$$

Depositar BNB en reservas requiere también depositar un valor equivalente a tokens BEP20. Esto se calcula con la ecuación:

$$tokensDeposited = tokenPool * \frac{bnbDeposited}{bnbPool}$$

5.2 Removing Liquidity

Providers can burn their liquidity tokens at any time to withdraw their proportionate share of BNB and BEP20 tokens from pools.

$$bnbWithdraw = bnbPool * \frac{amountBurned}{totalAmount}$$

$$tokensWithdraw = tokenPool * \frac{amountBurned}{totalAmount}$$

BNB and BEP20 tokens are withdrawn at the current exchange rate (reserve rate), not at the rate of your original investment. This means that some value may be lost due to market fluctuations and arbitrage.

Fees collected during trades are added to the total liquidity pools without minting new liquidity tokens. This results both, <bnbWithdraw> and <tokensWithdraw> which include a pro rate share of all fees collected since the liquidity was first added.

5.3 Liquidity Tokens

Exoswap Liquidity Tokens represent a contribution from liquidity providers to a BNB-BEP20 pair. They are BEP20 tokens and include a full implementation of transferring tokens as well as allowing tokens to be approved so that another third party in the chain is able to spend them.

This allows liquidity providers to sell their liquidity tokens or transfer them between accounts without removing liquidity from the funds. Liquidity tokens are specific to a single BNB \rightleftharpoons BEP20 exchange. There is no single unifying BEP20 token for this project.

6. Fee Structure

- BNB to BEP20 trades
 - 0.3% fee paid in BNB
- BEP20 to BNB trades
 - 0.3% fee paid in BEP20 tokens
- BEP20 to BEP20 trades
 - 0.3% fee paid in BEP20 tokens for BEP20 to BNB swap on input exchange
 - 0.3% fee paid in BNB for BNB to BEP20 swap on output exchange
 - Effectively 0.5991% fee on input BEP20

There is a 0.3% fee to switch between BNB and BEP20 tokens. This commission is distributed among the liquidity providers in proportion to their contribution to the liquidity reserves. Since BEP20 to BEP20 transactions include a BEP20 to BNB swap and a BNB to BEP20 swap, the fee is paid on both exchanges. There are no platform fees.

Swap fees are immediately deposited into liquidity reserves. Since the total reserves increase without adding additional share tokens, this increases the value of all share tokens equally. This works as a payment to liquidity providers that can be collected by burning shares.

7. Custom Fee

The contract launched to create an Exchange will have a fee for the implementer of 0.1% of all transactions made on its website where the widget is implemented or any ramp used as an api exchanged tokens in the Binance Smart Chain network. This function can be used as a B2B for both projects that launch their own token and companies that want to implement the Exchange function through a widget or api ramp in their business.

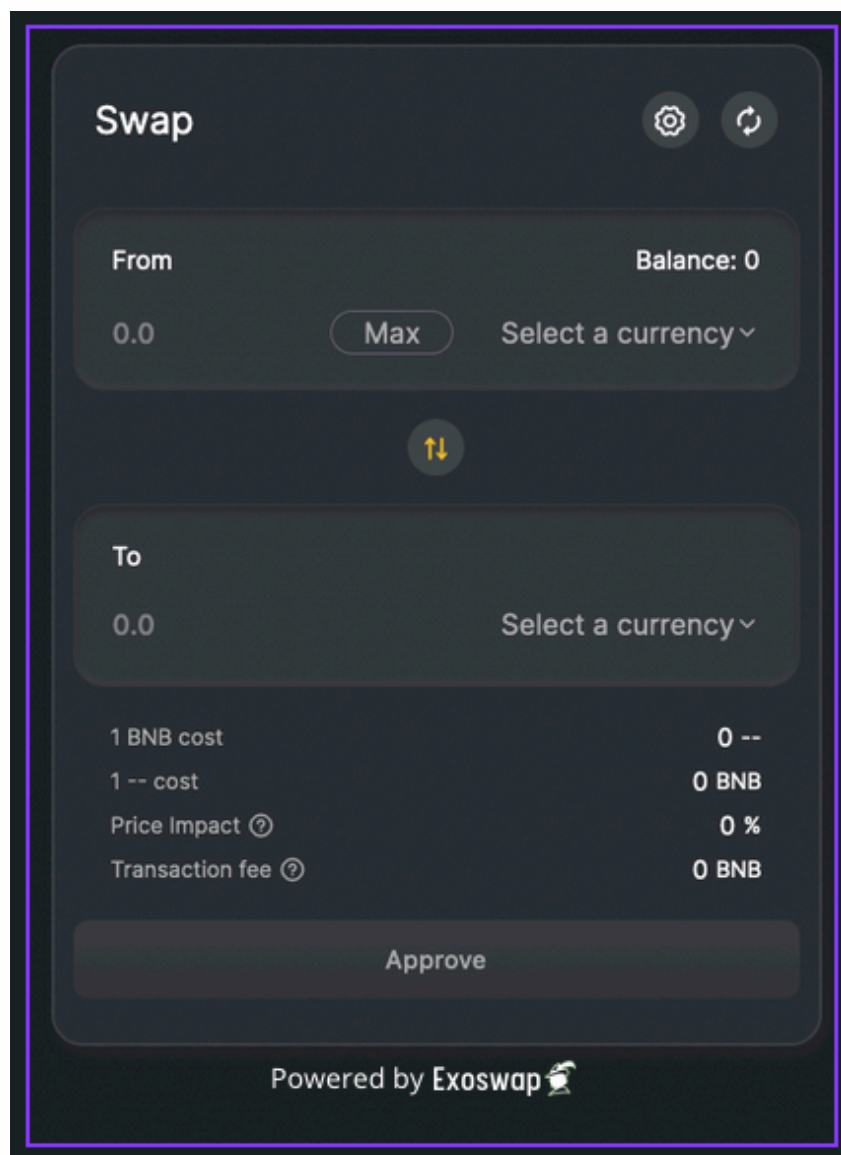
8. BEP20 to Exchange

Additional functions `tokenToExchangeSwap()` and `tokenToExchangeTransfer()` add to the flexibility of Exoswap. These functions convert BEP20 tokens to BNB and attempt a `bnbToTokenTransfer()` on a user input address. This allows BEP20 to BEP20 trades against custom Exoswap exchanges that don't come from the same lab, as long as they implement the proper interface. Custom exchanges can have different curves, managers, private liquidity pools, FOMO-based Ponzi schemes, or anything else you can think of.

9. Upgrades

Updating censorship resistant and decentralised smart contracts is challenging however Exoswap will work collaboratively with developers from the protocol community to stay on top. If an improved Exoswap skin is created in higher versions, a new factory contract can be implemented. Liquidity providers can choose to move to the new system or remain in the old one, which would be versions, for example (V1,V2,V3 and so on).

The tokenToExchange functions allow transactions with exchanges launched from different factories. This can be used for backward compatibility. Exchanges from BEP20 to BEP20 will be possible within versions that use the tokenToToken and tokenToExchange functions. However, in all versions only tokenToExchange will work. All updates are optional and previous ones compatible.



The image shows a dark-themed user interface for a swap function. At the top, the word "Swap" is displayed in a large, light-colored font. To its right are two circular icons: a gear for settings and a circular arrow for refresh. Below this, there are two main input sections. The first section, labeled "From", has a text input field containing "0.0", a "Max" button, and a dropdown menu labeled "Select a currency". To the right of the "From" section is a "Balance: 0" label. Between the "From" and "To" sections is a circular button with a yellow double-headed arrow. The second section, labeled "To", has a text input field containing "0.0" and a dropdown menu labeled "Select a currency". Below these sections is a table of costs and fees. The table has two columns: the first column lists the item, and the second column shows the value. The items are "1 BNB cost", "1 -- cost", "Price Impact", and "Transaction fee". The values are "0 --", "0 BNB", "0 %", and "0 BNB" respectively. Below the table is a large, light-colored button labeled "Approve". At the bottom of the interface, it says "Powered by Exoswap" followed by a small robot icon.

Item	Value
1 BNB cost	0 --
1 -- cost	0 BNB
Price Impact	0 %
Transaction fee	0 BNB

10. Exoswap technical market maker model and implementation

10.1 Overview

Consider a decentralised exchange that exchanges two tokens E and X. Let e and x be the number of tokens e and x , respectively, that the exchange currently reserves. The token exchange price is determined by the ratio of e and x , so that the product $e \times x$ is preserved. That is, when you sell Δe tokens, you will get Δx tokens such that $e \times x = (e + \Delta e) \times (x - \Delta x)$. Thus, the price ($\Delta e / \Delta x$) is a function of e/x . Specifically, when you trade Δe with Δx , the trading token reserves are updated as follows:

$$e' = e + \Delta e = (1 + \alpha)e = \frac{1}{1 - \beta}e$$

$$x' = x - \Delta x = \frac{1}{1 + \alpha}x = (1 - \beta)x$$

Where $\alpha = \frac{\Delta e}{e}$ and $\beta = \frac{\Delta x}{x}$. Also, we have:

$$\Delta e = \frac{\beta}{1 - \beta}e$$

$$\Delta x = \frac{\alpha}{1 + \alpha}x$$

Now consider a fee for each token exchange. Let $0 \leq \rho < 1$ be a fee, eg $\rho = 0.003$ for the 0.3% fee schedule.

$$e'\rho = e + \Delta e = (1 + \alpha)e = \frac{1 + \beta(\frac{1}{x} - 1)}{1 - \beta}e$$

$$x'\rho = x - \Delta x = \frac{1}{1 + \alpha}x = (1 - \beta)x$$

where $\alpha = \frac{\Delta e}{e}$, $\beta = \frac{\Delta x}{x}$, and $x = 1 - \rho$. Also, we have:

$$\Delta e = \frac{\beta}{1 - \beta} \cdot \frac{1}{x} \cdot e$$

$$\Delta x = \frac{\alpha x}{1 + \alpha} \cdot x$$

Note that we have the same formula above if there is no fee, i.e. $e = 1$. Also note that the product of e and x increases slightly for each trade due to the fee. That is, $e' p \times x' p > e \times x$ when $p > 0$, while $e' p \times x' p = e \times x$ when $p = 0$ (no charge).

In the contract implementation of this model, token E denotes BNB and token X denotes the token to trade. In addition, you can invest and divest, sharing exchange token reserves, which we will formalise later using the concept of liquidity. Since the implementation uses integer arithmetic, we also formally analyse the approximation error caused by integer rounding, showing that the error is bounded and doesn't lead to violation of critical properties denoted by the mathematical model.

10.2 Minting Liquidity

An investor can generate liquidity by depositing both BNB and tokens.

10.3 Add Liquidity

We formulate the mathematical definition of mint liquidity.

Definition a. `<add Liquidity spec>` takes as input $\Delta e > 0$ and updates the state as follows:

where $(e, t, l) \text{ addLiquidityspec } (\Delta e) \rightarrow (e', t', l')$

$$\begin{aligned} e' &= (1 + \alpha)e \\ t' &= (1 + \alpha)t \\ l' &= (1 + \alpha)l \end{aligned}$$

$$\text{and } \alpha = \frac{\Delta e}{e}$$

Here, an investor deposits both Δe BNB and $\Delta t = t' - t$ tokens and mints $\Delta l = l' - l$ liquidity. The invariant is that the ratio of $e : t : l$ is conserved and $k = e \times t$ increases, as formulated in the following theorem.

Theorem. Let $(e, t, l) \text{ <addliquidityspec> } (\Delta e) \rightarrow (e', t', l')$. Let $p = e \times t$ and $p' = e' \times t'$. Then, we have the following:

1. $e : t : l = e' : t' : l'$
2. $p < p'$
3. $\frac{p'}{p} = \left(\frac{l'}{l}\right)^2$

Definition b. <addLiquiditycode> takes as input an integer $\Delta e > 0 \in \mathbb{Z}$ and updates the state as follows:

$$(e, t, l) \in \mathbb{Z}^3 \text{ addliquiditycode } \Delta e \rightarrow (e'', t'', l'') \in \mathbb{Z}^3$$

$$\text{where } e'' = e + \Delta e = (1 + \alpha)e$$

$$t'' = t + \left[\frac{\Delta e * t}{e} \right] + 1 = [(1 + \alpha)t] + 1$$

$$l'' = l + \left[\frac{\Delta e * l}{e} \right] + 1 = [(1 + \alpha)l]$$

$$\text{and } \alpha = \frac{\Delta e}{e}$$

10.4 Burning Liquidity

An investor can withdraw their BNB and token deposit by burning their portion of liquidity. We formulate the mathematical definition of burning liquidity, being dual to minting liquidity.

Definition c. <removeliquidityspec> takes as input $0 < \Delta l < l$ and updates the state as follows:

$$\text{where } (e, t, l) \text{ removeliquidityspec } (\Delta l) \rightarrow (e', t', l')$$

$$e' = (1 - \alpha)e$$

$$t' = (1 - \alpha)t$$

$$l' = (1 - \alpha)l$$

$$\text{and } \alpha = \frac{\Delta l}{l}$$

Here, an investor burns Δl liquidity and withdraws $\Delta e = e - e'$ bnb and $\Delta t = t - t'$ tokens. The invariant is dual to that of coining liquidity.

Definition d. `<removeliquiditycode>` takes as input an integer $0 < \Delta l < l$ and updates the state as follows:

$$\text{where } (e, t, l) \in Z^3 \text{ } \text{removeliquiditycode}(\Delta l) \rightarrow (e'', t'', l'') \in Z^3$$

$$\text{where } e'' = e - \left\lfloor \frac{\Delta l * e}{l} \right\rfloor = [(l - \alpha)e]$$

$$t'' = t - \left\lfloor \frac{\Delta l * t}{l} \right\rfloor = [(l - \alpha)t]$$

$$l'' = l - \Delta l = (1 - \alpha)l$$

$$\text{and } \alpha = \frac{\Delta l}{l}$$

10.5 Token Price Calculation

We formalise the functions by calculating the current price of the token. Suppose there are two tokens E and X, and let e and x be the number of E and X tokens currently reserved by the exchange. We have two price calculation functions: `<getInputPrice>` and `<getOutputPrice>`. Given Δe , the `getInputPrice` function calculates how many X tokens (ie Δx) can be purchased by selling Δe . On the other hand, given Δx , the `<getOutputPrice>` function calculates how many tokens E (ie Δe) must be sold to buy Δx .

Please note that these functions don't update the status of the exchange.

Definition e. Let ρ be the trade rate. `<getInputPricespec>` takes as input $\Delta e > 0$, e , and x outputs Δx such that:

$$\text{getinputpricespec}(\Delta e)(e, x) = \Delta e = \frac{\alpha x}{1 + \alpha x} x$$

$$\text{where } \alpha = \frac{\Delta e}{e} \text{ and } x = 1 - \rho. \text{ Also, we have:}$$

$$e' = e + \Delta e = (1 + \alpha)e$$

$$x' = x + \Delta x = \frac{1}{1 + \alpha x} e$$

Definition f. Let ρ be the trade fee. `<getinputpricespec>` takes as input $\Delta x > 0$, e , and x , outputs Δx such that:

$$\text{getinputpricespec}(\Delta e)(e, x) = \Delta x = \frac{\alpha x}{1 + \alpha x} x$$

where $\alpha = \frac{\Delta e}{e}$ and $x = 1 - \rho$. Also, we have:

$$e' = e + \Delta e = (1 + \alpha) e$$

$$x' = x - \Delta x = \frac{1}{1 + \alpha x} x$$

Definition h. Let ρ be the trade fee. `<getinputpricecode>` takes a input $\Delta e > 0$, e , and $x \in \mathbf{z}$, and outputs $\Delta x \in \mathbf{Z}$ such that:

$$\text{getInputPricecode}(\Delta e)(e, x) = \Delta x = \left\lceil \frac{\alpha x}{1 + \alpha x} e \right\rceil$$

where $\alpha = \frac{\Delta e}{e}$ and $x = 1 - \rho$. Also, we have:

$$e'' = e + \Delta e = (1 + \alpha) e$$

$$x'' = x - \Delta x = \left\lfloor \frac{1}{1 + \Delta x} x \right\rfloor$$

In the contract implementation $\rho = 0.003$, and `<getinputpricecode>` $(\Delta e)(e, x)$ is implemented as follows:

$$(997 * \Delta e * x) / (1000 * e + 997 * \Delta e)$$

Definition i. Let p be the trade fee. $\text{getoutputpricespec}(\Delta x)$ takes a input $0 < \Delta x < x$, e , and x , and outputs Δe such that:

$$\text{getoutputpricespec}(\Delta x)(e, x) = \Delta e = \frac{\beta}{1 - \beta} \cdot \frac{1}{x} \cdot e$$

where $\beta \frac{\Delta x}{x} < 1$ and $x = 1 - p$. Also, we have:

$$e' = e + \Delta e = \frac{1 + \beta(\frac{1}{x} - 1)}{1 - \beta} \cdot e$$

$$x' = x - \Delta x = (1 - \beta)x$$

Definition j. Let p be the trade fee. $\text{getoutputpricecode}(\Delta x)$ takes as input $0 < \Delta x < x$, e , and $x \in \mathbf{Z}$, and outputs $\Delta e \in \mathbf{Z}$ such that:

$$\text{getoutputpricespec}(\Delta x)(e, x) = \Delta e = \left[\frac{\beta}{1 - \beta} \cdot \frac{1}{x} \cdot e \right] + 1$$

where $\beta \frac{\Delta x}{x} < 1$ and $x = 1 - p$. Also, we have:

$$e'' = e + \Delta e = \left[\frac{1 + \beta(\frac{1}{x} - 1)}{1 - \beta} \cdot e \right] + 1$$

$$x'' = x - \Delta x = (1 - \beta)x$$

In the contract implementation $p = 0.003$, and $\text{getinputpricecode}(\Delta x)(e, x)$ is implemented as follows:

$$(1000 * e * \Delta x) / (997 * (e - \Delta x)) + 1$$

10.6 Trading Tokens

Now we formalise the token exchange functions that update the exchange state. In this section, we present a formal specification of bnbToToken (including swap and transfer).

a. bnbToTokenspec

bnbToTokenspec takes and input $\Delta e (\Delta e > 0)$ and updates the state as follows:

$$(e, t, l) \text{ bnbToTokenspec } (\Delta e) \rightarrow (e', t', l)$$

where ;

$$e' = e + \Delta e$$

$$t' = t - \text{getinputpricespec } (\Delta e, e, t)$$

b. bnbToTokencode

bnbToTokencode takes and integer input $\Delta e (\Delta e > 0)$ and updates the state as follows:

$$(e, t, l) \text{ bnbToTokencode } (\Delta e) \rightarrow (e'', t'', l)$$

where;

$$e'' = e + \Delta e$$

$$t'' = t - \text{getinputpricecode } (\Delta e, e, t) = [t']$$

We present a formal specification of tokenToToken (including swap and transfer). Suppose there are two exchange contracts A and B, whose states are (e_A, t_A, l_A) and (e_B, t_B, l_B) respectively.

a. tokenToTokenspec takes an input $\Delta t_a (> 0)$ and updates the states as follows:

$$\{ (e_a, t_a, l_a), (e_b, t_b, l_b) \} \text{ tokenToToken } (\Delta t_a) \rightarrow \{ (e'_a, t'_a, l_a), (e'_b, t'_b, l_b) \}$$

where;

$$\begin{aligned} t'_a &= t_a + \Delta t_a \\ \Delta e_{\text{Aspec}} &= \text{getInputPricespec}(\Delta t_a, t_a, e_a) \\ e'_A &= e - \Delta e_{\text{Aspec}} \\ e'_B &= e_B + \Delta e_{\text{Aspec}} \\ \Delta t_{\text{Bspec}} &= \text{getInputPricespec}(\Delta e_{\text{Aspec}}, e_B, t_B) \\ t'_B &= t_B - \Delta t_{\text{Bspec}} \end{aligned}$$

11. Disclaimer

This whitepaper is for general information purposes only. It doesn't constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It shouldn't be relied upon for accounting, legal or tax advice or investment recommendations. This paper reflects current opinions of the authors and the opinions reflected in this document are a subject to change without being updated.